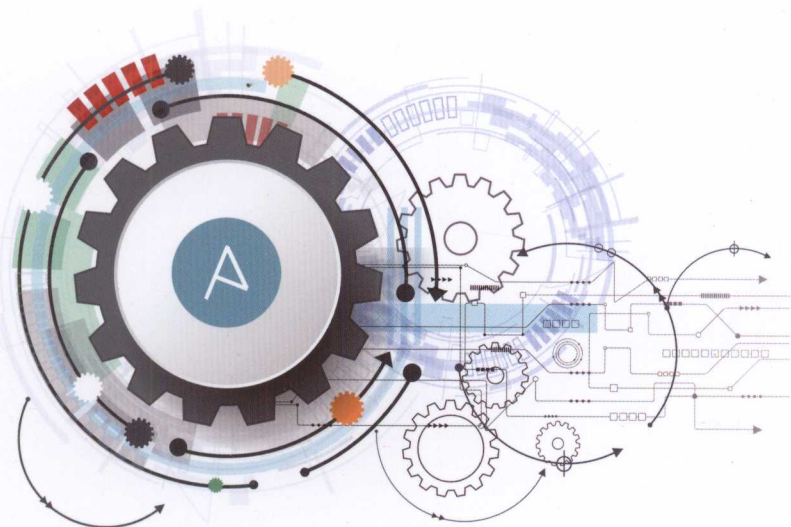


版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！

本书由资深运维人员联手打造，通过大量实例，详细讲解Ansible自动化运维方式与技巧。

从最基础的架构解析、安装配置，到典型应用场景与案例分析，作者分享了自己在工作中的实战经验，是掌握大规模集群运维管理的必备参考。



陈金窗 沈灿 刘政委 编著

*Ansible for Configuration Management and Automation*

# Ansible自动化运维 技术与最佳实践



机械工业出版社  
China Machine Press



## 作者简介

**陈金窗** 资深运维人员，曾就职于金山、姬慕石等互联网公司，目前在天翼云负责服务VIP客户运营。从事IT基础架构运维技术与管理近二十年，负责过多个大型IT运维项目，对大规模系统运维技术有深刻理解，乐于分享，组建了Ansible中国用户组QQ群，分享了大量文章和技术细节，极大地促进了Ansible技术的推广与应用。

**沈灿** 运维圈知名博主“灿哥”，曾就职于蓝讯、京东、Yottaa等互联网公司，对CDN和自动化运维技术有着丰富的经验，乐于分享，有很多技术文章和讲座广为流传。

**刘政委** 资深运维人员，从事大型在线游戏运维7年多，对系统集成、游戏自动化运维技术有丰富的经验，同时在社区分享了大量文章，广受好评。

## 图书在版编目 (CIP) 数据

Ansible 自动化运维: 技术与最佳实践 / 陈金窗, 沈灿, 刘政委编著. —北京: 机械工业出版社, 2016.2 (2016.12 重印)  
(实战)

ISBN 978-7-111-53115-9

I. A… II. ①陈… ②沈… ③刘… III. 程序开发工具 IV. TP311.52

中国版本图书馆 CIP 数据核字 (2016) 第 040638 号

本书由资深运维工程师联手打造, 通过大量实例, 详细讲解 Ansible 这个自动化运维工具的基础原理和使用技巧; 从基础的架构解析、安装配置, 到典型应用案例分析, 作者分享了自己在工作中的实战经验, 为各类运维操作、运维开发人员提供了翔实的指南。本书主要内容包括: Ansible 架构及安装, Ansible 组件、组件扩展、API, playbook 详解, 最佳实践案例分析, 用 ansible-vault 保护敏感数据, Ansible 与云计算的结合, 部署 Zabbix 组件、Haproxy + LAMP 架构, 以及 Ansible 在大数据环境的应用实战等。

## Ansible 自动化运维: 技术与最佳实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 殷 虹

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2016 年 12 月第 1 版第 2 次印刷

开 本: 186mm×240mm 1/16

印 张: 20.75

书 号: ISBN 978-7-111-53115-9

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

## Preface 前言

随着信息技术的迅速发展，形形色色的互联网应用已经成为我们日常生活不可分割的部分。云计算已经改变 IT 资源部署、配置和管理的方式，服务供应商向着“一切皆服务”交付模式努力。用户享受通过将基础设施扩展并作为服务使用带来的高效、便捷，服务供应商通过云生态环境能够向用户提供更高价值的服务。

这一切背后都有着庞大的 IT 系统做支撑，作为负责保障稳定运行的运维工作所面临的挑战越来越大。传统的人工运维方式已经无法满足业务的发展需求，需要从流程化、标准化、自动化去构建运维体系。随着 DevOps 运动的兴起，运维人员、研发人员、质量控制人员都从更大范围来看待自己的工作，打破运维、研发之间的壁垒，进行相互渗透、融合。DevOps 项目在数量和体量上持续增长，支撑持续集成、持续交付的自动化工具不断涌现。

Ansible 是 DevOps 项目基础支撑工具之一，是第一款实现读/写跨平台的“Infrastructure-as-code”工具，从系统管理者到开发者，都可使用 Ansible 自动化部署并维护整个应用的生命周期，实现持续交付。

Ansible 是 Github 上最热门的开源自动化工具之一，当前已经超过 1000 人为 Github 上的 Ansible 做过贡献。2013 年笔者创建的“Ansible 中国用户组”QQ 群（群号：142851673）也相当活跃，当前专业会员已超过 1000 人。

本书将带领读者探索 Ansible 自动化运维的神奇之旅，为运维工作节省时间、节约成本，并支持云环境应用部署。

## 读者对象

本书主要读者对象包括：

- IT 运维人员、系统管理员、企业网管。
- 运营开发人员、应用部署人员。
- 系统架构师。
- 大专院校的计算机专业学生。

## 主要内容

本书是笔者在多年的学习、研究、实践的基础上，对 Ansible 进行系统的总结和梳理，其中既包括对 Ansible 基础知识的详细讲解，又包括日常运维工作中典型应用场景的实践案例，还介绍 Ansible 业界丰富的进展和发展趋势。本书的实践案例和脚本，可以在实验和生产环境中针对本书描述的场景进行复制和使用。

本书的目标是介绍如何较好地使用 Ansible，从初始的命令行开始，到编写 playbooks，再到管理大型、复杂的环境，最后介绍如何构建自己的模块、编写插件扩展 Ansible 增加新的功能。对于新手来说，本书提供了关于自动化运维的具体操作实战。对有经验的维护人员来说，本书提供了如何把 Ansible 与具体应用相结合，讲解 Ansible 的最佳实践。对于产品专家来说，本书介绍了如何扩展 Ansible 自动化运维工具手段，讨论 Ansible 如何与其他系统的交互才能提供可满足最终用户需求的集成解决方案。

本书主体包括 14 章。各章可以独立阅读，但对于还没有大规模应用经验的新手，建议按照顺序、循序渐进阅读。

本书第 1、2、7、11 ~ 13 章由陈金窗编写，第 3 ~ 6、8 ~ 10、14 章、附录由沈灿编写，最后由刘政委进行校审。由于笔者的水平有限，编写时间仓促，且自动化运维方兴未艾，Ansible 当前仍处于快速发展之中，因此书中内容难免会出现一些错误或不准确的地方，恳请读者评判指正、不吝赐教。

## 致谢

首先感谢 Ansible 创始人 Michael DeHaan 和他的研发团队独具慧眼、发明创造了

功能强大、轻量级的自动化运维工具。同时感谢提供 Ansible 模块的所有第三方作者，是他们辛勤的劳动和乐于分享，才使得 Ansible 产生巨大威力，在他们身上闪烁着开源精神的绚丽光芒。

感谢机械工业出版社的编辑们一年来始终的支持、积极的鼓励、耐心的帮助，并逐字审阅、校正，才使本书的出版成为可能。

本书有一些内容参考了网络论坛、博客等，由于参考资料众多，有些时间久远无法了解确切出处，在此对热爱分享知识的网友表示深深的谢意。

最后，谨以此书献给我们最亲爱的家人和自己，以及众多热爱开源技术的网友们！

陈金窗

2016 年 2 月于北京



# 目 录 Contents

## 前 言

第 1 章 Ansible 架构及特点 .....	1
1.1 Ansible 软件及公司 .....	2
1.1.1 Ansible 应用领域 .....	3
1.1.2 Ansible 软件发布 .....	5
1.1.3 Ansible 公司服务 .....	8
1.2 Ansible 架构模式 .....	9
1.2.1 Ansible 管理方式 .....	10
1.2.2 Ansible 系统架构 .....	11
1.2.3 任务执行模式 .....	13
1.3 Ansible 特性 .....	14
1.3.1 Ansible 功能特性 .....	14
1.3.2 Ansible 与其他配置管理的对比 .....	21
1.4 Ansible 与 DevOps .....	22
1.5 本章小结 .....	26
第 2 章 Ansible 安装与配置 .....	27
2.1 Ansible 环境准备 .....	27
2.2 安装 Ansible .....	30

2.2.1	直接用源码安装	30
2.2.2	用包管理工具安装	32
2.3	配置运行环境	34
2.3.1	配置 Ansible 环境	34
2.3.2	使用公钥认证	36
2.3.3	配置 Linux 主机 SSH 无密码访问	36
2.4	Ansible 小试身手	38
2.4.1	主机连通性测试	38
2.4.2	在被管节点上批量执行命令	39
2.5	获取帮助信息	40
2.6	本章小结	42
<b>第 3 章</b>	<b>Ansible 组件介绍</b>	<b>43</b>
3.1	Ansible Inventory	43
3.2	Ansible Ad-Hoc 命令	49
3.3	Ansible playbook	56
3.4	Ansible facts	56
3.5	Ansible role	60
3.6	Ansible Galaxy	63
3.7	本章小结	63
<b>第 4 章</b>	<b>playbook 详解</b>	<b>64</b>
4.1	playbook 基本语法	64
4.2	playbook 变量与引用	70
4.3	playbook 循环	81
4.4	playbook lookups	91
4.5	playbook conditionals	96
4.6	Jinja2 filter	99
4.7	playbook 内置变量	102
4.8	本章小结	106

<b>第 5 章 Ansible 最佳实践</b>	107
5.1 优化 Ansible 速度	107
5.2 目录结构	113
5.3 定义多环境	115
5.4 灰度发布与检测	115
5.5 统一管理	116
5.6 使用 ansible-shell 交互命令行	116
5.7 本章小结	118
<b>第 6 章 扩展 Ansible 组件</b>	119
6.1 扩展 facts	119
6.2 扩展模块	125
6.3 callback 插件	130
6.4 lookup 插件	137
6.5 Jinja2 filter	139
6.6 本章小结	143
<b>第 7 章 用 ansible-vault 保护敏感数据</b>	144
7.1 了解 ansible-vault 如何保护数据	145
7.1.1 高级加密标准	145
7.1.2 ansible-vault 能够加密什么	145
7.2 使用 ansible-vault	146
7.2.1 创建加密数据文件	146
7.2.2 更新加密的数据文件	147
7.2.3 变更加密数据密钥	148
7.3 典型应用场景	148
7.3.1 实践场景 1: 保护 Ansible role 中的敏感数据	149
7.3.2 实践场景 2: 使用加密做用户认证	151
7.3.3 实践场景 3: 保护 Nginx 中的 SSL 密钥	152
7.4 本章小结	155

<b>第 8 章 Ansible 与云计算</b>	156
8.1 了解云平台管理流程	156
8.2 Ansible AWS 和 OpenStack	157
8.3 Ansible 与 Docker	162
8.4 Ansible Jenkins	165
8.5 本章小结	169
<b>第 9 章 部署 Zabbix 组件</b>	170
9.1 了解部署流程	170
9.2 编写业务 roles	171
9.3 安装部署	177
9.4 本章小结	179
<b>第 10 章 部署 HAProxy + LAMP 架构</b>	180
10.1 了解整体架构流程	180
10.2 编写业务 roles	181
10.3 配置部署以及测试	186
10.4 扩容与维护	188
10.5 本章小结	189
<b>第 11 章 大数据环境的应用实战</b>	190
11.1 某运营商大数据环境	191
11.2 准备大数据集群环境	192
11.2.1 安装操作系统	195
11.2.2 操作系统初始化	198
11.2.3 Ansible 无口令密钥执行环境	204
11.2.4 安装、配置 JDK	205
11.3 部署 Hadoop 集群	207
11.3.1 准备 Hadoop 基础角色	209
11.3.2 部署 NameNode 角色	219

11.3.3	部署资源管理器角色 .....	221
11.3.4	部署 DataNode 角色 .....	222
11.4	部署后 Hadoop 初始化与验证 .....	223
11.4.1	部署后初始化 .....	223
11.4.2	部署后 Hadoop 验证 .....	224
11.5	本章小结 .....	226
<b>第 12 章</b>	<b>Ansible 管理 Windows 系统 .....</b>	<b>227</b>
12.1	Ansible 管理 Windows 工作原理 .....	228
12.2	搭建 Ansible 管理工作组 Windows 环境 .....	229
12.2.1	安装、配置控制主机 .....	230
12.2.2	被管 Windows 主机配置 .....	230
12.2.3	配置资源清单 .....	232
12.2.4	测试被管 Windows 主机的连通性 .....	234
12.2.5	常见问题处理 .....	235
12.3	搭建 Ansible 管理活动目录 Windows 环境 .....	236
12.4	支持管理 Windows 模块 .....	239
12.5	常用 Windows 管理实例 .....	240
12.6	本章小结 .....	244
<b>第 13 章</b>	<b>网络自动化管理的应用实战 .....</b>	<b>246</b>
13.1	网络管理也自动化了 .....	246
13.2	Ansible 官方集成的网络角色 .....	249
13.3	生成配置文件及部署 .....	251
13.3.1	生成网络配置模板 .....	252
13.3.2	部署配置模板 .....	255
13.4	通过 SNMP 方式配置网络 .....	257
13.5	网络设备厂商提供接口实现自动化 .....	259
13.5.1	管理 Cisco NX-OS .....	259
13.5.2	管理 JUNOS .....	269



13.5.3 管理 Cumulus Linux .....	273
13.6 本章小结 .....	279
<b>第 14 章 Ansible API .....</b>	<b>280</b>
14.1 runner API .....	280
14.2 playbook API .....	283
14.3 使用 Flask 封装 Ansible API .....	286
14.4 使用 Celery 实现任务异步化 .....	290
14.5 使用 jQuery Ajax 异步请求 .....	297
14.6 本章小结 .....	300
<b>附录 A Ansible.cfg 配置文件参数详解 .....</b>	<b>301</b>
<b>附录 B YAML 与 Jinja .....</b>	<b>306</b>
<b>附录 C Ansible pull 模式 .....</b>	<b>312</b>
<b>附录 D SSH Forward 模式 .....</b>	<b>316</b>



## 第 1 章 Chapter 1

# Ansible 架构及特点

IT 行业的工作变得越来越有趣了，我们不再是把软件交付给客户，然后安装在单独的服务器上运行，我们都慢慢地变成了系统工程师。

我们现在部署应用软件的方式是通过服务串联起来，运行在一系列分布式的计算资源上并用各种不同的网络协议进行通信。常见的应用包括 Web 服务、应用服务、基于内存的缓存服务系统、任务队列、消息队列、SQL 数据库、NoSQL 数据存储、负载均衡等。

我们也需要确保采用合适的冗余，当故障发生时软件系统能够很好地处理、适应这些故障。另外有些辅助的服务需要部署、维护，例如日志管理、监控系统、分析系统，需要与第三方服务交互，如通过与 IaaS 接口交互来管理虚拟主机实例。

你可以用手动方式来搭建这些服务：安装服务器操作系统，SSH 登录每一台，安装软件包，编辑配置文件，等等。这种方式耗费大量时间还经常出错，特别是在做了 3 ~ 4 次之后，这枯燥重复的手工劳动是令人非常痛苦的。对于更复杂的任务，比如在你应用环境中搭建一个 OpenStack 云环境，由手工来操作会让人发疯。应有更好的方法。

如果你读到这里，你可能已经有了配置管理的思想，并考虑采用 Ansible 作为你的配置管理工具。无论你是一个开发人员想要把代码部署到生产环境，还是一个系统管理员寻找更好的自动化方法。我觉得 Ansible 对于这些问题都是很好的解决方案。

### 1.1 Ansible 软件及公司

IT 自动化配置管理最近 20 年获得了迅猛的发展，特别最近几年在移动互联、云计算、大数据、互联网 + 等大规模应用平台的需求推动下，涌现出一批成熟的大规模自动化运维工具。维基百科里列出了二十多个，其中 Puppet、Chef 和 Salt，以及 CFEngine、Vagrant 和 NixOS，大家都可能耳熟能详了。不过后起之秀 Ansible (<http://www.ansible.com/>) 的人气更高，已经是当今最常用的管理基础架构的开源管理工具之一。

从开源仓库 GitHub 上受到使用者、开发者的关注度、加星、贡献、评论（见表 1-1）可以看出，Ansible 的受欢迎程度的数据已经远远超过 Puppet、Chef、CFEngine、SaltStack。

表 1-1 GitHub 上开源自动化工具受关注程度信息表（截至 2015 年 8 月 30 日）

开源自动化配置工具	关注 (Watch)	加星 (Star)	复制 (Fork)	开始时间	评论数	贡献者
ansible/ansible	1 009	12 416	3 697	2012 年 2 月 5 日	15 821	1 146
puppetlabs/puppet	414	3 514	1 468	2005 年 4 月 10 日	20 618	394
saltstack/salt	451	5 573	2 370	2011 年 2 月 20 日	58 452	1 194
Chef/chef	338	3 794	1 554	2008 年 5 月 2 日	13 195	399
CFEngine/core	62	224	136	2007 年 12 月 30 日	12 544	73

Ansible 自从 2012 年 2 月发布以来，一直得到 Ansible 爱好者、用户、开发者的热情参与、持续贡献，如图 1-1 所示。

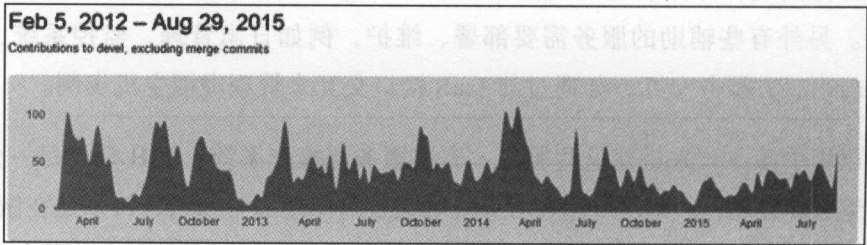


图 1-1 Ansible 受贡献者的支持趋势

Ansible 使用 Python 作为开发语言，巧妙地设计、实现了简单易用、功能强大的自动化管理工具。Ansible 由 Michael DeHaan 发起、开发、创建，他同时也是著名工具软件 Cobbler 与 Func 的开发者。Ansible 的第一个版本发布于 2012 年 2 月，目前下载量已经超过了 100 万。当前在 GitHub 上，它是排名前 10 位的 Python 项目，可以预见

Ansible 的发展不可限量。

Ansible 已经广泛应用于各种规模、各个领域的企业，包括 Rackspace、Twitter、Evernote、NASA、GoPro、Atlassian 等知名企业。

### 1.1.1 Ansible 应用领域

Ansible 的编排引擎可以出色地完成配置管理、流程控制、资源部署等多方面工作。与其他 IT 自动化产品相比较，Ansible 为你提供一种不需要安装客户端软件、管理简便、功能强大的基础架构配置、维护工具。

Ansible 基于 Python 语言实现，由 Paramiko 和 PyYAML 两个关键模块构建。Ansible 具有独特的设计理念：

- 安装部署过程特别简单，学习曲线很平坦。
- 管理主机便捷，支持多台主机并行管理。
- 避免在被管理主机上安装客户代理，打开额外端口，采用无代理方式，只是利用现有的 SSH 后台进程。
- 用于描述基础架构的语言无论对机器还是对人都是友好的。
- 关注安全，很容易对执行的内容进行审计、评估、重写。
- 能够立即管理远程被管理主机，不需要预先安装任何软件。
- 不仅仅支持 Python，可运行使用任何动态语言开发模块。
- 非 root 账户也可以使用。
- 成为最简单、易用的 IT 自动化系统。

在云计算时代的浪潮中，基础架构必须满足按需自动伸缩、按使用量计费的基本特性，IT 自动化运维软件就是最重要的必备工具之一。下面来看看几个关键的领域中取得的巨大的进展。

#### 1. 配置管理

配置管理领域已经涌现出多种工具，配置管理的目标就是确保被管理的主机尽可能快速、按照正确方式达到配置文件中描述的状态，这对管理 IT 环境至关重要。例如，在网站高峰时候需要扩展新的 Web 服务器，这需要一台由配置管理控制的机器能够快速就位，这也就是通常所说的代码化基础架构（Infrastructure as code），因为构建基础架构所有必须的代码都存储在源码控制系统中。这也是逐步引入对代码化基础架



构按照软件开发生命周期 (Software Development Lifecycle, SDLC) 方式进行管理, 这些包括辅助基础架构测试的工具具有 Ansible、CFEngine、Chef、Puppet、Salt 等, 基础架构测试工具具有 Serverspec、Test kitchen 等。

## 2. 服务即时开通

这个领域的工具主要是在数据中心、虚拟化环境、云计算中快速开通新的主机。几乎所有云计算的服务提供商都有相应的 API 接口, 这些自动化工具通过这些 API 接口能够快速地创建主机实例。对于基于 Linux 或最近快速发展的容器 (如 Docker、LXC), 越来越多的人开始采用自动化工具的方式来保证这些容器的开通。Ansible 在这些场景扮演了重要的角色。

## 3. 应用部署

这个领域的工具重点关注如何尽可能地零停机部署应用。许多单位已经采用滚动式部署 (rolling deployments) 或金丝雀部署 (canary deployments), Ansible 对这两种方式都支持。流水线式部署也是很常见的, 常见的工具包括 ThoughtWorks Go、Atlassian Bamboo、大量插件支持的 Jenkins 条, 都是比较优秀的。

## 4. 流程编排

流程编排主要是进行部署时候如何保证基础架构中的各种组件协调一致。例如, 在你 Web 服务器部署新的软件版本时候, 需要确保该 Web 服务器从负载均衡器上移出, 这是很常见的场景。这类工具有 Ansible、Mcollective、Salt、Serf、Chef 等。

## 5. 监控告警

监控告警工具已经发展到能够适应快速处理大规模服务器的环境。以前有成熟的 Nagios、Ganglia、Zenoss、Zabbix, 最新发展的有 Graphite、Sensu、Riemann 等, 都是相对不错的工具。

## 6. 日志记录

集中日志数据确保能够正确地收集跨系统和应用的日志, 同时能够按照规则进行智能过滤、根本原因分析、告警等。常见的工具有 Logstash-Kibana、SumoLogic、Rsyslog 等。

在上面关键的六个领域中, Ansible 能够非常完美地完成前面四个领域的工作。通



过使用 Ansible，无论是系统管理员、运维团队、基础架构管理员、开发者，或者其他任何需要基础架构自动化者都可以从中受益。本书目的就是介绍如何构建健壮的 IT 基础架构自动化运维系统。

### Ansible 软件创始人：Michael DeHaan

2012 年 2 月，曾在 Red Hat 开发 Cobbler 和 Func、又在 Puppet 工作过的 Michael DeHaan 看到了 IT 自动化领域的机会：Linux 管理员不得不用好几类工具来应付不同的工作场景，如配置管理用 Puppet 或 Chef，部署时要用 Fabric 或 Capistrano，还要用 Func 或 mCollective 处理其他任务，总之，太复杂了。同时，多节点部署却没有处理得很好的工具，而在云计算和大规模互联网的基础设施里，这恰恰是最有意思的问题。

一天，DeHaan 在自己的沙发上开始用 Python 开发一个新工具，他的目标是：极为易用，连他自己都很想用；任何人可以在几分钟之内学会并使用。经过短短的 6 个月，第一个版本的工具诞生了，这就是 Ansible。

由于 DeHaan 在运维圈已经很有名气，Ansible 发布后很快流行起来。这期间，Fedora 的 Seth Vidal(yum 作者)采用并在 4 月份发表了 High Scalability，都非常关键。

这之后，DeHaan 还参与了 OpenStack 的开发，但在用 Puppet 自动化管理 OpenStack 的过程中不断撞墙。这时候，Ansible 在 GitHub 上火了起来。很快他决定成立公司——AnsibleWorks。2013 年 8 月公司获得 600 万投资，后来改名为 Ansible 公司。

Ansible 只依赖 SSH，无需在远程机器上安装代理，极为容易上手。Hacker News 上有人称之为 shell scripting++，很到位。

### 1.1.2 Ansible 软件发布

Ansible 公司负责 Ansible 开源软件的维护、管理，是 Ansible 软件发展的最大贡献者。Ansible 开发团队非常高效，软件发布周期大约是 2 个月发布一个新版本。由于发布周期如此之短，轻微的 bug 通常是在下一个版本中得到修补，而不是对稳定版本发布新补丁。重大的 bug 经评估后，如果确实需要将会发布对稳定版本的补丁，但这种情况很少出现。

Ansible 项目重要目标之一是向后兼容，因此这些实例不用修改也能在将来的版本中很好地运行。Ansible 每个主要版本代号都是 Van Halen 乐队的一首曲子。在编写本书时，最新稳定版是 2015 年 7 月 26 日发布的 1.9.2 版本，代号为“Dancing In the Street”。1.9.2 是对之前 1.9.0/1.9.1 的小版本升级，主要修复一些安全、功能方面的 bug。

编写本书时，2.0 版本已经在开发中，但还没最后确定内容和发布时间，主要将改进以下内容。

## 1. 变更功能

- 引入新的 block/rescue/always 指令符，允许执行任务块和异常处理的语义。
- 扩展新的 plugin 策略，可以在每个 play 中控制任务执行的过程，默认时与以前一样。
- 改进异常处理，现在你将可以得到更多的详细解析信息，将会更新一般的异常处理和显示。
- 任务中 include 将在执行者进行评判，最终的结果与之前一样，但现在可以包含动态的 include 和选项。
- 新版本中将可以使用更多动态的 include 的重要特性以“with\_”开头的循环。
- Callback、connection、lookup 的插件 API 略有变化，新版中有的需要修改才能工作。
- Callbacks 现在与活动目录一起，不需要复制，只需要添加到 ansible.cfg 的白名单中。
- 许多 API 已经变更，虽然现在运行的不能直接在新版本中运行，但新的 API 更容易使用、测试。
- 设置将更有继承性，你在 play、block 或 role 中设置的参数将由容器自动继承，这样将可以在所有层级上设置，以前需要手工处理这些代码。
- 模板代码现在保持 bool 或数字类型，而不是转换成字符串，如果你要用以前的方式，只要把值用引号引起来，将会是按照字符串方式处理；如果是 null 的情况，输出的结果将会是一个空字符串。
- 增加了 refresh\_inventory 元语句，强制在 play 中重新读取资源清单。
- vars 现在可以在 play、block、role 和 task 不同级别中设置。
- 在 yaml 中设置的空变量或 null 变量将不再转换成空字符串，它们将保留 None 的值；如果要保留以前的处理方式，只需要在配置文件中设置 null\_representation 或环境变量 ANSIBLE\_NULL\_REPRESENTATION 中进行设置。

## 2. 改进模块

对少量模块进行更新, 包括 `ec2_ami_search` (`ec2_ami_find`)、`quantum_network` (`os_network`)、`glance_image`、`nova_compute` (`os_server`)、`quantum_floating_ip` (`os_floating_ip`), 括号内是对应的新版本模块。

## 3. 新增

新增多达 79 个支持模块, 主要是增强对云计算环境的支持, 特别是对 Amazon、CloudStack、OpenStack、VmWare 等主流云平台的支持, 还有些是增强对应用 WebFaction、Win\_IIS、Zabbix 等的支持。

### Ansible 名称的来历

最早是厄休拉·勒古恩 (Ursula K. Le Guin) 在 1966 年的小说《罗卡农的星球》(Rocannon's World) 中创造了 Ansible 这个词, 用以表示一种能在浩瀚宇宙中即时通信的装置。这在她后来的作品中也得到了沿用, 并很快传播到了其他科幻作家的作品之中, Ansible 往往作为一种速度比光速还快的虚构通信工具。最出名的案例也许是奥森·斯科特·卡德 (Orson Scott Card) 的《安德的游戏》(Ender's Game), 其中地球两次遭遇过虫族的进攻, 国际舰队认为必须在世界各地寻找天资聪颖的孩童, 将把他们塑造成舰队指挥官, 使人类在与虫族的战斗中占领先机, 并得到存活希望。安德使用 Ansible 装置实时远程指挥前线的舰队, 这是相距数光年作战的唯一可行方式。

Michael DeHaan 想用这个词来比喻控制远端大量的服务器。

那么, 勒古恩又是怎么想到这个单词的呢? 在 2001 年 Usenet 的一个帖子里, 戴夫·古德曼宣称勒古恩曾经告诉他 “Ansible” 是从 “answerable” (可以应答) 演变来的, 她后来发现把这个词的字母换一下顺序就变成了 “lesbian” (女同性恋者), 这一点也使她觉得相当有趣。

## 4. 扩展

扩展了新的资源清单, 支持 CloudStack、Fleetctl、Openvz、Proxmox、Serf 等环境。

详见 Ansible 官方信息发布网站 <https://github.com/ansible/ansible/blob/devel/>

CHANGELOG.md 说明，或 <http://www.ansible.cn/forum.php?mod=viewthread&tid=334> (部分已翻译)。

### 1.1.3 Ansible 公司服务

Ansible 既可以是指软件开源的名称，也可以指运营开源项目的公司名称。Michael DeHaan 是 Ansible 软件的发起者、创始人，Ansible 公司前技术总监 (CTO)。为了避免混淆，我们在后面涉及 Ansible 都指 Ansible 软件，指公司的时候都用 Ansible 公司表示。

Ansible 公司是负责 Ansible 软件开发、开源社区的管理，确保 Ansible 软件工具适应 IT 自动化的需求。同时 Ansible 公司在开源 Ansible 软件基础之上，开发了基于 Web 界面友好的 Ansible Tower 专有 IT 自动化工具。提供 Ansible 软件推广、咨询、培训服务，为支持 Ansible 发展的可持续提供保证。Ansible 的服务团队成员都是经过精心挑选、久经沙场、经验丰富的 IT 专业人士，包括系统管理员、软件开发、咨询服务精英、自动化运维专家。无论是公共的还是专有的领域，他们在各种规模系统上都取得过成功。团队的所有成员都是高级的 Ansible 用户，都为开源产品贡献过代码，是社区的活跃成员。他们提供如下一些服务。

#### 1. Ansible 健康检查服务

如果已经使用 Ansible 有段时间了，但你想要确保符合最佳实践标准；如果你的上线日期已经很近，想要在上限截至日期之前一切准备就绪，都可以寻求“健康检查服务”。服务团队将评估你 Ansible 和 Ansible Tower 管理的内容和配置，用 Ansible 最佳实践来验证你的系统，并为你将来的项目提供建议。有如下服务：

- 审查你当前的 Ansible 配置是否遵循最佳实践。
- 建议改进你 Ansible 的实现。
- 对在审查过程中发现的问题将按照目标进行培训。

#### 2. 协助重大迁移服务

发现 Ansible 是很易用的产品，但现在已经部署了 Puppet、Chef、CFEngine 或其他定制化的解决方案。要从现在的配置管理、部署方案、创建 Ansible 与之对应，这需要做大迁移服务工作，这项工作包括：

- 评估：与你原来使用工具的专家一起分析当前工具管理的内容，设计一个迁移计划。



- 实现：根据评估的结果，按照推荐的方式进行实施，用 Ansible 工具创建相应的管理功能。

根据客户的需求，Ansible 咨询团队将交付部分或全部迁移服务，使运行在 Ansible 维护工具中。

Ansible 团队还将提供一份文档化的解决方案，一种可行的迁移方式，探寻“train the trainer”方式，让你的团队将来能够自己可以进行迁移。

### 3. 推进 IT 自动化项目

推进自动服务是一项能够加快建立起自动化很好方法，这些协议将提供：

- 为你的团队培训 Ansible、Ansible Tower 的基础使用和高级知识。
- 安装、配置 Ansible 和 Ansible Tower。
- Ansible 导师将在你的流程中部署关键组件。

### 4. 客户咨询服务

Ansible 公司可以提供个性化客户服务，包括从项目启动、健康检查、重要迁移到最后完成设计目标过程中大大小小所有事情。

可能交付的内容包括：定制化模块、Jinja2 模板、动态资产库脚本、角色和 playbook，能够帮助你解决挑战的问题，加速任何 Ansible 项目的推进。

### 5. Ansible 培训服务

Ansible 官方网站已经提供了快速入门的视频短篇教程，帮助你了解 Ansible。这个短篇介绍了 Ansible 作为一款强大的自动化 IT 运维、流程编排的工具，有哪些优势，这也是为你开始自动化项目提供必备的基础知识。

Ansible 提供老师讲解的课程，包括动手学习环境、实际问题处理、现场建设性问题解答。也致力于根据你的个性需求开发特定裁剪的课程，个性化的培训也可以根据要达到的目标签订专业的服务协议。

## 1.2 Ansible 架构模式

Ansible 维护模式通常由控制机和被管机组成。控制机是用来安装 Ansible 工具软



件、执行维护指令的服务器或工作站，是 Ansible 维护的核心。被管机是运行业务服务的服务器，由控制机通过 SSH 来进行管理。

### 1.2.1 Ansible 管理方式

Ansible 是一个模型驱动的配置管理器，支持多节点发布、远程任务执行。默认使用 SSH 进行远程连接。无需在被管节点上安装附加软件，可使用各种编程语言进行扩展。

Ansible 总体系统构成如图 1-2 所示。Ansible 管理系统由控制主机和一组被管节点组成。控制主机直接通过 SSH 控制被管节点，被管节点通过 Ansible 的资源清单 (inventory) 来进行分组管理。

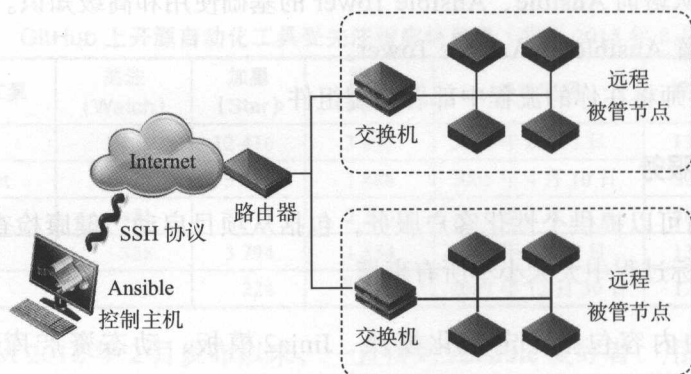


图 1-2 Ansible 总体系统构成

Ansible 如何进行配置管理呢？图 1-3 给出了实例，这是 Ansible 用剧本 (playbook) 方式对 3 台运行 Nginx 服务的 Ubuntu 服务器进行配置管理。她编写一个叫 `webservers.yml` 的 Ansible 脚本。在 Ansible 中，这个脚本称为 `playbook`。在 `playbook` 中描述了包含被管节点的 `hosts` 和对这些 `hosts` 按照顺序执行的任务列表 (`task`)。

在此例子中，`hosts` 包括 `web1`、`web2`、`web3`，任务列表包括如下过程：

- 安装 Nginx (`Install Nginx`)。
- 创建 Nginx 配置文件 (`/etc/nginx/nginx.conf`)。
- 基于安全证书 SSH 方式拷贝配置文件，重启 Nginx 服务。
- 确保 Nginx 服务处于启动状态。

然后在 Ansible 系统的控制主机上执行：

```
$ ansible-playbook webservers.yml
```

Ansible 将会通过 SSH 连接并行地在 web1、web2、web3 上面安装、配置、运行 Nginx 服务。

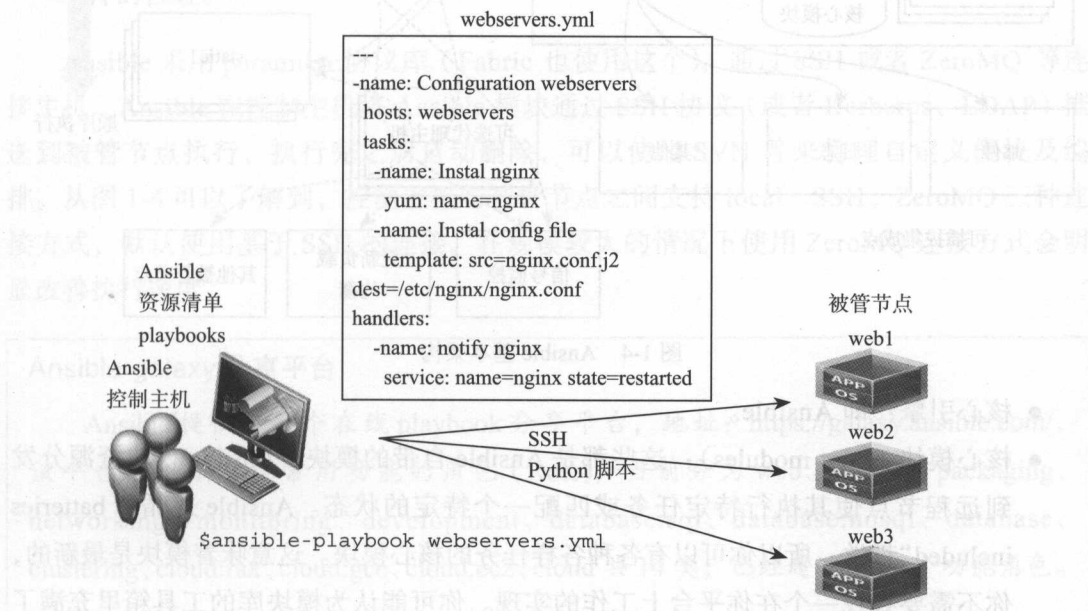


图 1-3 运行 Ansible 的 playbook 对 3 台 Web 服务器进行配置

本书后面的章节将会对这里涉及如何编写 playbook，以及 webservers.yml 中各个字段的含义、工作内容做深入的讲解。

### 1.2.2 Ansible 系统架构

Ansible 集合了众多优秀运维工具 (Puppet、Cfengine、Chef、Func、Fabric) 的优点，实现了批量系统配置、批量程序部署、批量运行命令等功能。Ansible 是基于模块工作的，本身没有批量部署的能力。真正具有批量部署的是 Ansible 所运行的模块，Ansible 只是提供一种框架。Ansible 的基本架构见图 1-4，用户通过 Ansible 编排引擎操作公有云 / 私有云或 CMDB (配置管理数据库) 中的主机。

从图 1-4 可以看到，Ansible 由以下各部分组成：

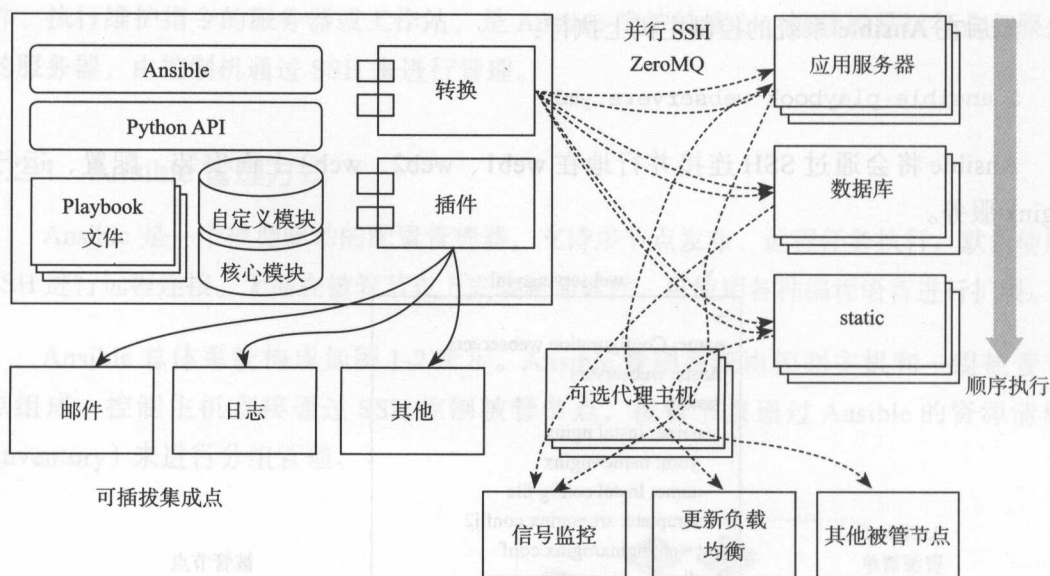


图 1-4 Ansible 基本架构

- 核心引擎：即 Ansible。
- 核心模块（core modules）：这些都是 Ansible 自带的模块，Ansible 模块资源分发到远程节点使其执行特定任务或匹配一个特定的状态。Ansible 遵循 “batteries included” 哲学，所以你可以有各种各样任务的核心模块。这意味着模块是最新的，你不需要寻找一个在你平台上工作的实现。你可能认为模块库的工具箱里充满了有用的系统管理工具，和 playbook 一起构建自动化运维系统的基础源材料。
- 自定义模块（custom modules）：如果核心模块不足以完成某种功能，可以添加自定义模块。
- 插件（plugins）：完成模块功能的补充，借助于插件完成记录日志、邮件等功能。
- 剧本（playbook）：定义 Ansible 任务的配置文件，可以将多个任务定义在一个剧本中，由 Ansible 自动执行，剧本执行支持多个任务，可以由控制主机运行多个任务，同时对多台远程主机进行管理。
- playbook 是 Ansible 的配置、部署和编排语言，可以描述一个你想要的远程系统执行策略，或一组步骤的一般过程。如果 Ansible 模块作为你的工作室工具，playbook 是你的设计方案。在基本层面上，剧本可用于管理配置和部署远程机器。在更高级的应用中，可以序列多层应用及滚动更新，并可以把动作委托给其他主机，与监控服务器和负载均衡器交互。

- 连接插件 (connector plugins): Ansible 基于连接插件连接到各个主机上, 负责和被管节点实现通信。虽然 Ansible 是使用 SSH 连接到各被管节点, 但它还支持其他的连接方法, 所以需要有连接插件。
- 主机清单 (host inventory): 定义 Ansible 管理的主机策略, 默认是在 Ansible 的 hosts 配置文件中定义被管节点, 同时也支持自定义动态主机清单和指定配置文件的位置。

Ansible 采用 paramiko 协议库 (Fabric 也使用这个), 通过 SSH 或者 ZeroMQ 等连接主机。Ansible 在控制主机将 Ansible 模块通过 SSH 协议 (或者 Kerberos、LDAP) 推送到被管节点执行, 执行完之后自动删除, 可以使用 SVN 等来管理自定义模块及编排。从图 1-4 可以了解到, 控制主机与被管节点之间支持 local、SSH、ZeroMQ 三种连接方式, 默认使用基于 SSH 的连接。在规模较大的情况下使用 ZeroMQ 连接方式会明显改善执行速度。

#### Ansible-galaxy 分享平台

Ansible 提供了一个在线 playbook 分享平台, 地址: <https://galaxy.ansible.com/>, 该平台汇聚了各类常用功能的角色 (Role), 当前分为 web、system、packaging、networking、monitoring、development、database:sql、database:nosql、database、clustering、cloud:rax、cloud:gce、cloud:ec2、cloud 共 14 类, 已经超过 4000 个功能角色。

根据需要找到适合你的角色后, 只需要使用命令 “ansible-galaxy install+ 作者 ID. 角色包名称” 就可以安装到本地, 比如想安装 debops 提供的 Nginx 安装与配置的角色, 直接运行 “ansible-galaxy install debops.nginx” 即可安装到本地, 该角色的详细地址为: <https://galaxy.ansible.com/list#/roles/1580>。

### 1.2.3 任务执行模式

Ansible 系统由控制主机对被管节点的操作方式可分为两类, 即 ad-hoc 和 playbook:

- ad-hoc 模式使用单个模块, 支持批量执行单条命令。
- playbook 模式是 Ansible 主要管理方式, 也是 Ansible 功能强大的关键所在。playbook 通过多个 task 集合完成一类功能, 如 Web 服务的安装部署、数据库服务器的批量备份等。可以简单地把 playbook 理解为通过组合多条 ad-hoc 操作的配置文件。



在 Ansible 系统内部又是如何执行的呢？Ansible 执行过程如图 1-5 所示。

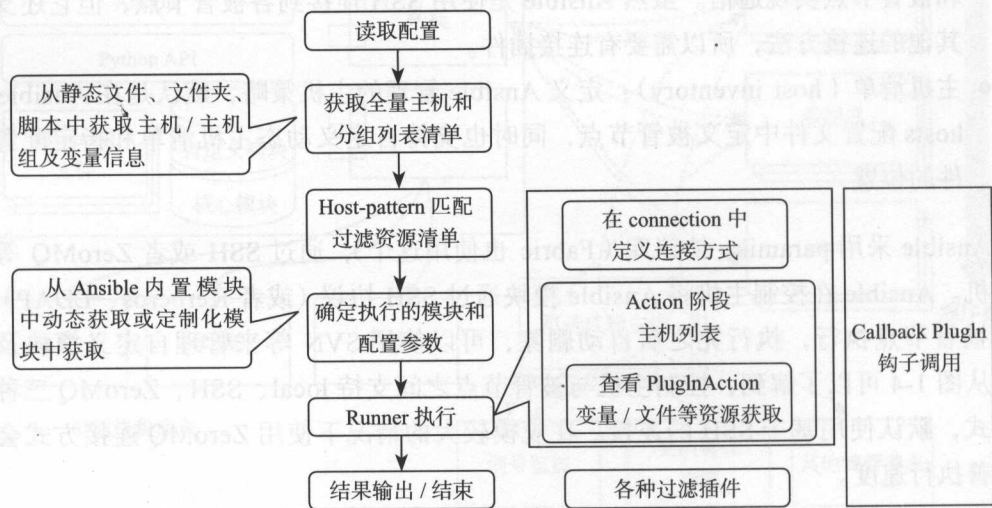


图 1-5 Ansible 执行过程

### 1.3 Ansible 特性

Ansible 是基于一致性、安全性、高可靠性设计的轻量级自动化工具，具有功能强大、部署便捷、描述清晰等特性。对于管理员、开发者、IT 经理等都容易上手，学习曲线较低，能够快速理解、掌握 Ansible 的自动化体系，满足不同技术级别的用户需求。

同时 Ansible 是一款满足当代大规模、复杂环境的 IT 基础架构自动化管理的工具。Ansible 相对与其他自动化解决方案，在核心能力上效率更高。也很好地解决了统一配置、统一部署、流程编排等复杂的 IT 自动化管理问题。下面就介绍其功能特性以及其他工具的对比。

#### 1.3.1 Ansible 功能特性

从功能上看，Ansible 可以实现以下目标：

- 应用代码自动化部署。
- 系统管理配置自动化。
- 支持持续交付自动化。
- 支持云计算、大数据平台（如 AWS、OpenStack、CloudStack、VMWare 等）环境。



- 轻量级，无需在客户端安装 agent，更新时只需在控制机上进行一次更新即可。
- 批量任务执行可以写成脚本，不用分发到远程就可以执行。
- 使用 Python 编写，维护更简单，Ruby 语法过于复杂。
- 支持非 root 用户管理操作，支持 sudo。

实现上述目标是很有挑战性的工作，需要满足健壮性、易于管理的架构，这在应用维度一直没能很好地解决，因为管理工具不能对被管环境施加额外的影响，而 Ansible 很好地解决了这些问题。Ansible 作为优秀管理工具具备以下一些特点。

### 1. 语法简单、易读

Ansible 的配置管理脚本 playbook 语法基于 YAML，它是一种可读性高、用来表达数据序列的格式标准。YAML 参考了 XML、C、Python、Perl 以及电子邮件格式 RFC2822 等其他多种语言，可读性好、交互性强，使用了实现语言的数据类型，有一个一致的信息模型，可以基于流方式来处理，具有表达能力强、扩展性好等特点。

可以把 playbook 称为“可执行的文档”，就像 README 文件一样描述你需要部署软件的指令，由于这些代码是可以直接执行的，这些指令永远不会过时。

### 2. 不需要在被管节点安装客户端软件

Ansible 项目管理系统的核心是通过 SSH 来连接被管节点，可以使用 paramiko（一个 Python 库）或使用原生的 OpenSSH（带参数 -c ssh）。当使用 SSH 客户端连接时候，默认的 OpenSSH 连接是可以重用的。无论哪种方式，SSH 都是作为一种传输方式，不是作为一种执行的 Shell。

模块是包含一些参数的 Ansible 小程序，通过 SCP 或 SFTP 传送到远端被管机器的临时目录中进行执行，然后再删除。这些模块通过标准的输出方式返回 JSON 格式的数据，这些返回的数据通过控制主机上的 Ansible 程序进行处理。

这种方式能管理非常多远程活动，并且只需要很小的交互流量。模块按照“幂等资源”的方式进行管理，也就是发起多次执行也不会给服务带来负面响应。它不是简单的命令或脚本，例如一个模块可以确定应该安装某个特定版本的软件包，但如果这个系统已经有合适的工作状态，就不会再执行任何命令。这种方式有如下优点：

- 提高网络安全：由于不需要在远端被管节点上运行 agent，因此 Ansible 遭受攻

击的机会很少。只需要运行 OpenSSH 一个后台进程，它在全球范围内是最严格审查的程序之一。Ansible 知道做好加密是一项极其艰难的事情，因此没有考虑使用自己的后台进程和认证方式，而是依赖可用的最安全的远程管理系统。OpenSSH 极其广泛地在各种系统中使用，有时非常轻量级。一旦 OpenSSH 出现安全问题，将会非常快速地推出补丁修复。当我们认为可能编写一个安全的 OpenSSH 实现时，我们也一直跟踪远程拓展在这一应用领域的类似工具，期望尽量避免出现类似问题。

- 可以使用非 root 用户访问（可用 sudo）：Ansible 的 playbook 能够用任何用户账号登录远程系统，可以用初始连接的用户来运行程序，也可以使用 sudo 切换到其他用户（包括 root）。如果需要可以直接用 root 登录，sudo 可以需要口令，也可以不需要口令，当管理的一些系统根本不能使用 root 登录的时候，或者不允许 root 登录但可以通过 sudo 切换到 root 的，这些方式是非常合适的。这里有个例子，一个没有超级权限的用户能够用 Ansible 管理自己 home 目录下的内容，即使在这个系统上没有 root 或 sudo 权限。甚至当使用 sudo 时，文件传输是没有限制的。Ansible 也包含一个兼容 sudo 的文件传输工具，内容是按照正常的 SFTP 方式传输的，Ansible 会把文件复制到授权的位置。通过比较源文件和目标文件的校验和方式，Ansible 能够智能地识别文件是否需要传输。
- 限制传输潜在敏感数据：Ansible 总是尽量把最少的数据传输到被管节点，由于控制服务器是逻辑控制中心，只有远端被管节点必要的变量才送给它。例如，有一个全局变量叫 foo，只有在远端被管节点中明确在资源或模板中使用它，否则它不会发送到该远端节点。这种 Ansible 的推送方式，只给远端节点需要看到的最少数据。类似地，Ansible 没有包含客户文件服务器的实现，只通过 SFTP、SCP、Rsync（基于 SSH）传输文件，并且只有 playbook 需要的文件才传输。结果是被管节点请求的文件或模板可以提供，但敏感数据不会提供给。也就是不能给一台远程主机查看将要提供给其他被管节点的数据。这使得 Ansible 适用于带有敏感数据的环境，包括社会科学工作、健康医疗、政府政务等应用。
- （Credential Segregation 凭证隔离）：Ansible 适用于不同用户有不同安全级别的环境。可以定义管理主机的通用变量，然后使用自己的凭证访问远端节点。例如，允许研发工程师管理开发的服务器，QA 工程师管理 QA 的服务器，系统管理员管理生产环境服务器，不存在研发工程师把代码推送到生产环境的风险。
- 去中心化：由于 Ansible 的 playbook 需要用户的凭证来执行，不存在中心点，只

是通过访问配置内容、管理软件链条，不需要访问 SSH 密钥，就可以接管建立“僵死网络”，避免使它成为攻击的目标。用户可以加密自己的密钥，不需与任何人使用相同密码。在可选的管理形式中，访问授权的软件资源都有可能导致部署系统自动化。要使用 Ansible，用户需要两个条件：能够访问到资源库，具有管理远端节点的凭证。当使用锁定的密钥（甚至是密码）时，不存在暴露安全漏洞中心被攻击点，也不允许从远端硬件来访问管理服务器。

- 没有“管理的管理程序”的问题：许多配置管理解决方案的主要问题之一是“管理的管理”，为了开始管理服务器必须先远端节点上安装软件。当更新管理软件时候，通常各种 agent 需要先更新（许多系统是无法自己自动更新的）。有时会产生管理服务器与 agent 软件版本兼容性问题，或 agent 与运行语言版本问题，Ansible 通过基于 SSH 传输模块避免了这类问题。SSH 二进制代码已经是 OS 的一部分了，并且是每个主流操作系统的核心。不需要任何 agent，也就不存在 agent 以任何方式崩溃的问题，因此服务器存在的安全风险是非常小的。
- 管理服务器的可扩展性：由于 Ansible 使用推送方式来管理远端节点（当然，也很容易配置成远端节点查询更新的方式），因此 Ansible 对于“羊群效应”的管理问题是有免疫能力的。在其他的一些解决方案中，管理 agent 周期性检查会对管理服务造成破坏，经常会超过管理服务器的运行负荷，导致需要对管理服务器进行横向或纵向扩展。频繁管理服务器需要为远端节点消耗昂贵的计算资源。为了解决这个问题，Ansible 通过推送的方式，通过配置每次推送节点数量来做限制。均衡了远程节点需要的管理服务器的负载能力，可以让计算系统负载更加均衡，即使个人笔记本电脑也可以作为 Ansible 系统的控制服务器。
- 资源的利用：在 Ansible 不管理远端节点时，这些远端节点什么也不做，也就是没有后台进程在消耗 CPU、内存。曾经报道过，某些应用服务器在唤醒配置窗口时会产生明显的降低性能，某些 agent 可能会占用超过 400MB 的内存。在虚拟化环境中，这些资源的消耗可能会快速叠加，就会增加对硬件的支出。Ansible 能最大限度让你的关键性能负荷使用所有的计算资源，你也可以选用间隔方式运行管理程序，也不存在由于内存泄漏或 agent 软件崩溃导致无法管理远端节点的问题。
- 防火墙的友好性和确定性：不像某些基于消息总线的系统，Ansible 不需要长时间在控制节点与被管主机之间建立连接。在生产环境，这种长连接可能受限于防火墙，防火墙一般不喜欢长连接，Ansible 很好地规避了这个问题。当管理服务与应用之间的连接被断开又没法重置的时候，就只能等到重启 agent 端服务了，

Ansible 没有这个问题。没有严格意义上的无代理系统，Ansible 也避免在这些架构中的其他问题，获得确定性的响应。并不是只有出现响应节点才管理，对于你要管理但无法达到的就静默了。如果一个节点宕机，运行 Ansible 时你就会知道，将会返回一个失效信息，这个你可能忽略或修复。这对于你给所有节点或部分子集部署软件更新，是很重要的信息，知道哪些节点无法连接是非常重要的，而对“发布-订阅”架构通常是没有提供这些信息的。

### 3. 基于推送 (Push) 方式

有些配置管理系统（如 Chef、Puppet、Saltstack 等）是使用代理模式的，它们默认基于拉 (pull) 方式。被管节点上安装代理后，代理服务将周期性检查中心控制主机的服务，并从控制主机上取来配置信息。被管节点的变更过程大致如下：

- 1) 修改配置管理脚本。
- 2) 把修改好的管理配置脚本推送到中心控制主机上。
- 3) 被管节点的代理服务周期性触发检查。
- 4) 被管节点连接配置管理的控制主机。
- 5) 被管节点下载新的配置管理脚本。
- 6) 被管节点在本地执行配置管理脚本，被管节点状态相应地变化。

而 Ansible 默认基于推送 (push) 方式，管理过程如下：

- 1) 增加或修改 playbook。
- 2) 执行新的 playbook。
- 3) 控制节点的 Ansible 连接被管节点并执行修改被管节点状态的模块。

一旦你执行 `ansible-playbook` 命令，Ansible 就会连接到远程的被管节点并按照脚本要求执行。

基于推送方式有很大的优势，你能控制什么时候让远程被管节点发生变更，不需要等到被管节点上周期性的时间。拉方式的支持者声称拉方式具有管理服务器规模数



量的优势，新被管节点随时可以在线添加。但是 Ansible 已经在管理超过上万台节点规模的系统，很方便地动态增减或删除被管节点。

当然，如果你一定要使用拉方式，Ansible 官方也已经支持这种拉方式，可以使用 `ansible-pull` 这个工具。

#### 4. 方便管理小规模场景

Ansible 能够管理成千上万台主机，但如何管理缩小规模的场景呢？其实，用 Ansible 也很容易配置小规模集群甚至单台主机，你只要简单编写一个 `playbook` 即可。Ansible 遵循艾伦·凯（Alan Kay）的名言：“简单的东西应该简单，复杂的东西才有可能成功（Simple things should be simple, complex things should be possible）”。

#### 5. 大量内置模块

使用 Ansible 能够在被管节点上执行任何 `shell` 命令，但是 Ansible 真正的威力在于内置大量的模块。使用这些模块可以完成如软件包安装、重启服务、拷贝配置文件等操作。Ansible 模块是声明式的（*declarative*），你只需使用这些模块描述被管节点期望达到的状态。例如，你可以用 `user` 模块对用户授权，确保系统有一个 `deploy` 用户，并且属于 `web` 组，代码如下：

```
user: name=deploy group=web
```

Ansible 内置模块都是等幂性的（*idempotent*）。这就是如果系统中没有用户 `deploy`，Ansible 将会创建一个 `deploy` 账号；如果这个账号已经存在，则 Ansible 什么也不做。等幂性对于自动化维护是非常重要的特性，这样在被管节点上多次执行 Ansible 的 `playbook` 也能达到同样效果。相对于直接执行操作系统脚本的方式是巨大的改进，操作系统脚本再执行一次可能就会产生不同的、非预期的结果。

#### 什么是收敛性

配置管理的资料中经常提到“收敛”这个概念，在配置管理中收敛最切确的说法是在 Mark Burgess 以及他的 CFEngine 配置管理中所写的。

如果一个配置管理系统是收敛的，那么这个系统可以多次运行，都达到最终期望的状态，每次执行只是更加接近而已。



这个收敛的思想在 Ansible 中不完全适用，Ansible 没有在被管节点上多次运行的概念，Ansible 在模块中实现了这种方式，一次执行 playbook 就会确保每台被管节点都能达到期望的状态。

“幂等性”和“收敛性”是有差别的，而且有非常重大的差异。幂等性是指如果系统已经处于期望的状态，则对系统什么也不操作。而收敛性是指系统不需要变更或操作的特性。最终的结果可能一样，但 CFEngine 相对于幂等性更强调收敛性，并且在自动化系统中内嵌了大量的逻辑，避免执行不必要的操作。

## 6. 非常轻量级的抽象层

有些配置管理工具通过一个抽象层，可以把相同的脚本在不同的操作系统上管理。例如，安装软件包有 yum、apt 工具，对于配置管理工具就可能抽象成 package。

Ansible 不同于这种方式，你需要使用 apt 模块来安装基于 apt 软件管理的系统，用 yum 模块来安装基于 yum 软件管理的系统。

这听起来也许不太方便，在实践中我们发现这使得 Ansible 更方便使用。Ansible 不需要再学新知识来屏蔽不同操作系统的差异性、新的抽象环境。这相对减少了 Ansible 要求的知识领域。这样，你不需要了解太多就可以编写 playbook 了。

如果你确实需要，你也可以在 Ansible 的 playbook 中对不同被管节点的操作系统编写不同的动作脚本。但不太建议这么做，建议编写针对特定操作系统的 playbook，如 CentOS、Ubuntu。

Ansible 社区中最主要的重用是模块，模块都是比较小的，与特定操作系统相关，具有良好的定义，容易实现，方便共享。Ansible 项目是非常开放的，经常接纳社区贡献的模块代码。

Ansible 的 playbook 不是真正地在不同上下文之间能够重用，本书后面讲到的 roles 方式整合 playbook 是更好的重用方式，大量在线的 roles 资源库收集在 Ansible Galaxy 中。现在已经超过 4000 个。

工作实践中，每个单位配置的服务器有所差异，应最好为自己的单位编写 playbook，而不是尝试重用通用的 playbook。查看其他人的 playbook 主要是学习人家如何做到的。

目标是提供面对广泛的自动化挑战如何获得大型生产力的优势。当 Ansible 提供更强大的生产力逐步替代其他许多核心性能的自动化解决方案时，它也在寻求解决其他还没解决的 IT 挑战，如何将复杂多层级工作流程清晰化、清楚统一地配置 OS、在单一框架下进行应用程序的部署。

### 1.3.2 Ansible 与其他配置管理的对比

当前几款与 Ansible 功能类似的主流配置管理软件有 Chef、Puppet、SaltStack 等，这里只对各个软件的技术特性做个简单对比，其中不针对各个软件的性能作比较。具体内容见表 1-2。

表 1-2 Ansible、Puppet、SaltStack 技术特性比较

项目	Puppet	SaltStack	Ansible
开发语言	Ruby	Python	Python
是否有客户端	有	有	无
是否支持二次开发	不支持	支持	支持
服务器与远程机器是否相互验证	是	是	是
服务器与远程机器通信是否加密	是，标准 SSL 协议	是，使用 AES 加密	是，使用 OpenSSH
平台支持	支持 AIX、BSD、HP-UX、Linux、Mac OS X、Solaris、Windows	支持 BSD、Linux、Mac OS X、Solaris、Windows	支持 AIX、BSD、HP-UX、Linux、Mac OS X、Solaris
是否提供 Web UI	提供	提供	提供，不过是商业版本
配置文件格式	Ruby 语法格式	YAML	YAML
命令行执行	不支持，但可通过配置模块实现	支持	支持

与其他的自动化工具相比较，Ansible 不需要安装管理客户端就能轻松地管理、配置你的基础架构。它们各自的优缺点见表 1-3。

表 1-3 Ansible、Puppet、Chef、SaltStack 的优缺点

产品	优势	劣势	成本
Ansible	<ul style="list-style-type: none"> <li>模块可以用任何语言开发</li> <li>被管节点不需要安装代理软件</li> <li>有 Web 管理界面，可以配置用户、组、资源清单和执行 playbook</li> <li>安装、运行极其简单</li> </ul>	<ul style="list-style-type: none"> <li>对被管节点是 Windows 有管理待加强</li> <li>Web 管理界面是内置 Ansible 的一部分</li> <li>需要导入资源清单</li> </ul>	<ul style="list-style-type: none"> <li>Ansible 开源版本是免费的</li> <li>Ansible Tower 小于 10 台时被管节点是免费</li> <li>超过 10 台之后每年每台需要支付 \$100 ~ \$250 的支持服务费用</li> </ul>

(续)

产品	优势	劣势	成本
Puppet	<ul style="list-style-type: none"><li>● 模块由 Ruby 或 Ruby 子集编写</li><li>● push 命令能够立即触发变更</li><li>● Web 界面生成处理报表、资源清单、实时节点管理</li><li>● 在代理运行端进行详细、深入的报告和对节点进行配置</li></ul>	<ul style="list-style-type: none"><li>● 需要学习 Puppet 的 DSL 或 Ruby</li><li>● 安装过程缺少错误检查和产生错误报表</li></ul>	<ul style="list-style-type: none"><li>● 开源版本是免费的</li><li>● Puppet 企业版需要每年每台花费约 \$100</li></ul>
SaltStack	<ul style="list-style-type: none"><li>● 状态文件可以用简单的 YAML 配置模板或复杂的 Python/PyDSL 脚本</li><li>● 与客户端通信可以基于 SSH 或在被管节点安装代理</li><li>● Web 界面可以看到运行的工作、minion 状态、事件日志，可以在客户端执行命令</li><li>● 扩展能力极强</li></ul>	<ul style="list-style-type: none"><li>● Web 界面相对于竞争产品还不太完整、稳定</li><li>● 缺乏生成深度报告的能力</li></ul>	<ul style="list-style-type: none"><li>● 开源软件是免费的</li><li>● SaltStack 企业版每年每个节点花费约 \$150，随着数量增加将有优惠折扣</li></ul>
Chef	<ul style="list-style-type: none"><li>● 可充分使用 Ruby 的强大功能</li><li>● 集中基于 JSON 的 data bags 在运行时产生变量</li><li>● Web 界面可以搜索节点清单，查看活动节点，分配 Cookbooks、roles 和节点</li></ul>	<ul style="list-style-type: none"><li>● 需要 Ruby 的编程知识</li><li>● 当前缺少 push 功能的命令</li><li>● 文档有时不太清晰</li></ul>	<ul style="list-style-type: none"><li>● 开源版本是免费的</li><li>● 5 台企业版的 Chef 是免费的，20 台以内费用每月是 \$120，50 台以内每月是 \$300，100 台以内每月是 \$600，等等</li></ul>

这些自动化管理软件都具备强大的功能、灵活的系统管理和状态配置，都提供丰富的模板及 API，对云计算平台、大数据都有很好的支持。

Puppet、Chef、SaltStack 设置比较复杂，有代理和服务，有远程被管节点需要长期运行管理进程，而且很多术语和概念都是自己一套。Ansible 相对于其他产品要简单得多，本质上混合了声明式和命令式。不仅适用于大规模的 IT 环境，而且对于一些规模较小的环境（20 ~ 50 台机器）Ansible 也是很实用的。

## 1.4 Ansible 与 DevOps

本节简要介绍下 DevOps(Development 和 Operations 的组合)。DevOps 是一组过程、方法与系统的统称，用于促进软件开发（应用程序 / 软件工程）、技术运营和质量保障 (QA) 部门之间的沟通、协作与整合，如图 1-6 所示。

DevOps 概念的引入能对产品交付、测试、功能开发和维护（包括常见的“热补丁”）起到意义深远的影响。在缺乏 DevOps 能力的组织中，开发与运营之间存在着信息“鸿沟”，例如运营人员要求更好的可靠性和安全性，开发人员则希望基础设施响应

更快，而业务人员的需求则是更快地将更多的特性发布给最终用户使用。这种信息鸿沟就是最常出问题的地方。

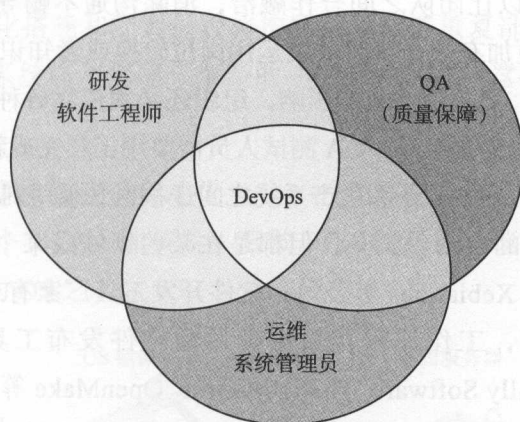


图 1-6 DevOps 的涉及范围

需要频繁交付的单位更需要具有 DevOps 能力，大量互联网在线移动应用随时根据用户的反馈进行迭代开发，持续改进用户体验，这需要能够支撑业务部门每天可能都有几次、几十次部署的需求。这种能力也称为持续部署，并且经常与精益创业方法联系起来。下面几个因素促使引入 DevOps：

- 使用敏捷或其他软件开发过程与方法。
- 业务负责人要求加快产品交付的速度。
- 虚拟化和云计算基础设施日益普遍。
- 数据中心自动化技术和配置管理工具的普及。

DevOps 将使开发团队与运营团队之间更具协作性、更高效的关系。由于团队间协作关系的改善，整个组织的效率因此得到提升，伴随频繁变化而来的生产环境的风险也能得到降低。团队之间相互融合，运维从开发流程的计划阶段就参与进来，运维团队就能了解将要开发的产品，同时可以在让产品开发设计初期就考虑后期运维的需求。

DevOps 成功的关键有如下因素：

- **文化建设。**首先要调动开发和运营部门之间的协作，鼓励运营人员采纳软件开发方法，利用云计算基础设施来完成测试和代码部署。其次在软件开发、测试、质量保障、集成、预生产和生产部署等方面必须打散小团队，更好地整合开发和运



营人员。例如，在讨论运营解决方案或扰乱事后评估报告时应该邀请开发人员加入。相反地，应该邀请运营人员列席开发人员规划会议。让交叉组合的工作模式成为制度，可以让团队之间合作融洽，消除沟通不畅导致的延误或疏忽，使 DevOps 的推进更加有效。有时可以运用岗位轮换或者知识共享的方法。

- **自动化工具支撑。**要超越文化的影响，组织还必须依靠各种 DevOps 工具。例如，开发人员编写代码需要工具，QA 测试人员需要用工具完成新版软件的部署，环境准备、将新代码在测试系统和生产系统之间迁移也必须用到云资源调度工具。

现在已经有了大量的自动化工具，但都是在某些领域或某个领域做得好。运营工具厂商有 BMC、CA 和 XebiaLabs 等公司，软件开发工具厂家有 IBM、Electric Cloud、Serena Software 等公司，工作流程、架构设计和软件发布工具的专业供应商包括 Atlassian、CollabNet、Rally Software、ThoughtWorks、OpenMake 等公司。评估供应商时，除了对基本功能考察之外，还需要考虑这些公司随时可能会被并购，其产品可用性和未来发展也会因此受影响。

尽管企业 IT 环境中的应用各种各样，但其操作系统、基础组件还是有很多共性的。手动安装操作系统的过程通常需要小时级才能交付。并且安装系统的可靠性完全依赖于管理人员的工作经验、技术能力、对复杂过程操作的精确程度。然后管理员需要对这些系统一遍又一遍地重复同样的过程，如图 1-7 所示。

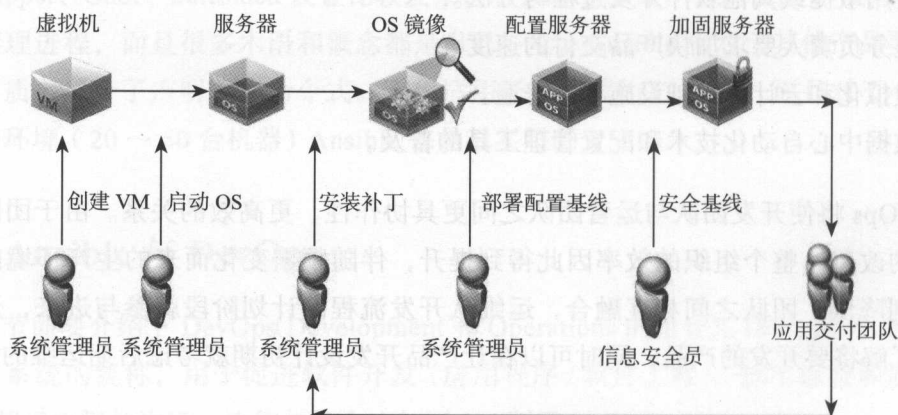


图 1-7 传统工厂运维方式

也可能让其他团队一起参与进来搭建系统，在应用交付团队能够工作之前，信息安全团队需要验证是否达到安全基线。需要接触系统的团队越多，实现的时间就越长，



也就更加可能延时和增加错误方式。

服务器和系统还比较好理解、容易自动化。尽管你当前 OS 搭建过程有很多需要与基础环境交互，自动化搭建和配置过程将按照需要能够重复部署 OS 镜像、合适的工具，管理这些建好的系统将与创建一个新的一样简单，这样这些系统就会总是看起来差不多。

一旦操作系统的搭建和管理自动化之后，把这些自动化推广到其他团队就相对容易了。通常客户搭建一个 OS，如研发、测试、QA 团队运行他们的应用时，能够确保总是运行在配置合适的 OS 之上。

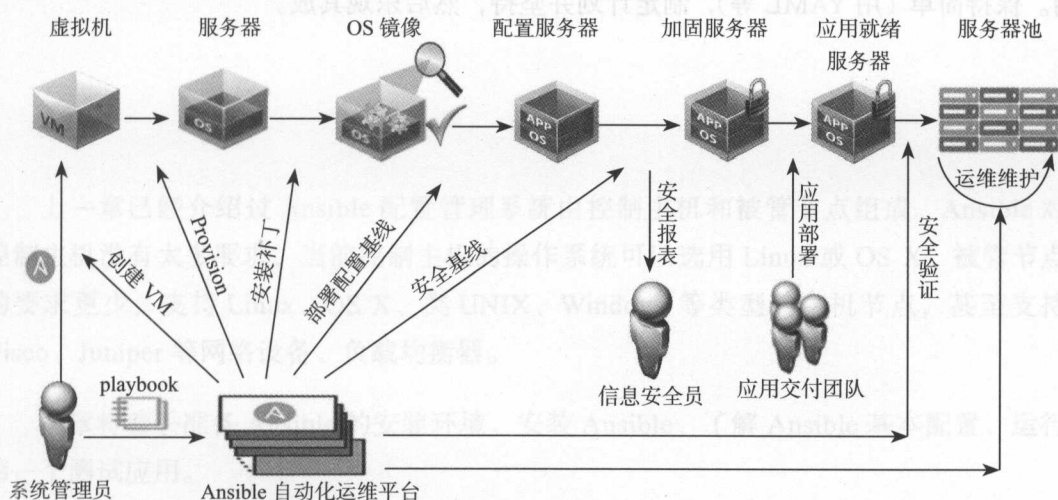


图 1-8 Ansible 自动化运维方式

但是，仅仅搭建系统，就忽略了更困难部分的问题了：如何保持它们整个生命周期中处于更新状态，如何保证当前的基本需求是正确的？这再一次需要采用自动化工具来管理它们之间的差异。

过去传统虚拟机（VM）生命周期管理方法是一个主要问题，需要独立的流程来维护存在的虚拟机，许多交付工具都对在线运行系统的更新、变更缺乏有效手段。

幸运的是，借助服务器自动化搭建和维护过程，可大大减少或完全消除服务器手工交付的过程，使用 Ansible 作为自动化工具，使用相同的 playbook 来搭建系统，就能够保证创建出来的系统是一样的。

甚至在基础架构层应用，也可以自动创建、交付、管理，将节省大量的时间，为单位的扩展团队提供额外的好处。

## 1.5 本章小结

Ansible 关键的想法是，开发一个好的自动化系统，能认识到计算机是一组而不只是一个个分开的机器，也就是所谓“多层编排”。建模过程与建模状态同样重要。不按传统配置管理依赖定制代理架构的思路，避免了证书交换，以及反向解析 DNS 和 NTP 的问题。默认可插拔，人人都可以很容易地贡献，因此 Ansible 获得了广泛的参与和采用。保持简单（用 YAML 等），制定计划并坚持，然后乐观其成。



在运维工作中，我们经常需要部署大量的应用，如果每个应用都通过手工部署，那么部署的时间成本是非常高的。Ansible 提供了一种基于 YAML 的声明式配置管理工具，通过编写 Playbook 文件，可以实现对多台主机的批量部署和管理。Ansible 的架构非常简单，它由一个中央的 Ansible Controller 和多个被管理的 Hosts 组成。Controller 通过 SSH 协议与 Hosts 进行通信，并执行各种操作。Ansible 的模块化设计使得用户可以轻松地扩展其功能，通过编写自定义的 Modules 和 Plugins 来满足特定的需求。此外，Ansible 还支持通过 Playbooks 来定义复杂的部署流程，从而实现自动化部署和配置管理。通过 Ansible，运维人员可以大大提高部署效率，减少人为错误，并确保系统的一致性和稳定性。

## Ansible 安装与配置

上一章已经介绍过 Ansible 配置管理系统由控制主机和被管节点组成。Ansible 对控制主机没有太多要求，当前控制主机的操作系统可以选用 Linux 或 OS X。被管节点的要求更少，支持 Linux、OS X、类 UNIX、Windows 等类型的主机节点，甚至支持 Cisco、Juniper 等网络设备、负载均衡器。

本章将着手准备 Ansible 的安装环境、安装 Ansible、了解 Ansible 基本配置、运行第一个测试应用。

### 2.1 Ansible 环境准备

安装前的准备工作包括思考如下一些内容：

- 从哪里获取安装软件包？
- 需要安装哪些支撑软件？
- 安装哪个版本的软件？
- 控制主机需要做哪些准备工作？
- 被管节点需要准备什么条件？

下面就分头讲讲。

### 1. 从 GitHub 获取 Ansible

Ansible 项目源码放在 GitHub 上管理，可以从 <https://github.com/ansible/ansible> 获取代码。也可以从这里了解到 Ansible 项目的最新进展，跟踪 Ansible 最新的思想和 bug 的处理情况。

### 2. 不需要安装支撑软件

Ansible 默认是基于 SSH 协议进行通信的。安装 Ansible 之后，控制主机不需要启动或运行任何 Ansible 的后台进程，也不需要数据库。只要在一台控制主机上安装好，就可以通过这台主机管理一组远程节点。在远程被管节点上，同样也不需要安装、运行任何 Ansible 特有的软件。这样如果 Ansible 版本需要升级，只需升级控制主机，不涉及被管节点。

### 3. 选择开发版本

由于 Ansible 基于简单的源码运行，不必在被管节点上安装特有软件，因此很多用户会使用 Ansible 的开发版本。

Ansible 一般每两个月出一个发行版本。小 bug 一般在下一个发行版本中修复，并在稳定分支中用 backport（将一个软件的补丁应用到比此补丁所对应的版本更老的版本的行为）方式增加补丁。大 bug 将会在必要时出更新维护版本，这种情况很少出现。

希望使用 Ansible 最新的版本，并且使用的操作系统是 RHEL、CentOS、Fedora、Debian、Ubuntu 等，我们建议使用系统软件包管理工具安装。

另有一种选择是通过 pip 工具安装，pip 是一个安装和管理 Python 包的工具。

如果希望使用开发版本，需要使用、测试最新的功能特性，可以直接下载源码、运行，源码程序不需要安装过程，直接可以使用。

### 4. 准备控制主机

只要主机上安装了 Python 2.6 或以上版本，就可以运行 Ansible 配置工具，Windows 环境系统当前还不能作为控制主机。控制主机需要有下面这些组件：

- Python 2.6 或以上
- paramiko 模块

- PyYAML
- Jinja2
- httpLib2

控制主机的系统可以是各种类 UNIX 操作系统，如 Red Hat、Debian、CentOS、OS X、BSD 等各种版本。

### 5. 查看被管节点

被管节点如果是类 UNIX 系统，则需要安装 Python 2.4 或以上版本。但如果版本低于 Python 2.5，则需要额外安装一个模块：python-simplejson 模块。Ansible 的 raw 模块和 script 模块是不需要 python-simplejson 模块支持的。从技术上讲，可以通过 Ansible 的 raw 模块安装 python-simplejson，之后就可以使用 Ansible 的所有功能了。

被管节点如果是 Windows，则需要有 PowerShell 3.0 并授权远程管理。

#### 管理开启 SELinux 环境

如果被管节点上启用了 SELinux，需要安装 libselinux-python，这样才可使用 Ansible 中与 copy/file/template 相关的函数。可以通过 Ansible 的 yum 模块在需要的托管节点上安装 libselinux-python。

#### 关于 Python 版本

Python 3 与 Python 2 是稍有不同语言，而大多数 Python 程序还不能在 Python 3 中正确运行。而一些 Linux 发行版（Gentoo、Arch）默认没有安装 Python 2.X。在这些系统上，需要专门安装 Python 2.X 解释器，并把资源清单（inventory）中的 ansible\_python\_interpreter 变量设置为该 Python 2.X。当然，也可以使用 raw 模块在被管节点上远程安装 Python 2.X。

RHEL、CentOS、Fedora、Ubuntu 等发行版都默认安装了 2.X 的解释器，几乎所有的 UNIX 系统也预安装了。

为了方便读者理解，笔者通过虚拟化环境部署了两组业务功能服务器来进行演示。笔者的操作系统版本为 CentOS 7.0，自带 Python 2.7.5，将采用下一节介绍的 yum 方式安装。相关服务器信息如表 2-1 所示（CPU 核数及 Web 根目录的差异化便于演示生成动态配置）。



表 2-1 业务环境表

角色	主机名	IP 地址	组名	CPU (核数)	Web 根目录
控制主机	ansiblecontrol	192.168.1.100	---	---	---
被管节点	web1	192.168.1.111	webservers	2	/website
被管节点	web2	192.168.1.112	webservers	2	/website

本章后续的安装过程，也根据这个环境来进行。

## 2.2 安装 Ansible

Ansible 的安装方式非常灵活，满足各种环境部署的需求。一般可以直接用源码进行安装，也可用操作系统软件包管理工具进行安装，下面分别介绍。

### 2.2.1 直接用源码安装

#### 1. 从 GitHub 源码库安装方式

很容易从 Ansible 项目的 GitHub 源码库提取出来安装，运行 Ansible 不需要 root 权限，也不依赖于其他软件，没有后台进程运行，不需要数据库支撑。不少社区用户直接使用 Ansible 的开发版本，这样可以利用最新的功能特性，也方便对项目进行测试。由于被管节点不需要安装任何软件，及时跟进、更新 Ansible 开发版相对于其他开源项目要容易得多。

从源码安装的过程如下：

##### 1) 提取 Ansible 源代码：

```
$ git clone git://github.com/ansible/ansible.git --recursive
$ cd ./ansible
$ source ./hacking/env-setup
```

如果想要安装过程中减少告警 / 错误信息输出，可以在安装时加上 -q 参数：

```
$ source ./hacking/env-setup -q
```

##### 2) 如果系统没有安装过 pip，先安装对应 Python 版本的 pip：

```
$ sudo easy_install pip
```

3) 安装 Ansible 控制主机需要的 Python 模块:

```
$ sudo pip install paramiko PyYAML Jinja2 httpplib2 six
```

4) 当更新 Ansible 版本时,不但要更新 git 的源码树,还要更新 git 中指向 Ansible 自身的模块,称为 submodules:

```
$ git pull --rebase
$ git submodule update --init --recursive
```

5) 一旦运行 env-setup 脚本,就意味着 Ansible 从源码中运行起来了。默认的资源清单 inventory 文件是 /etc/ansible/hosts,清单文件 inventory 可以指定其他位置:

```
.. code-block:: bash
$ echo "127.0.0.1" > ~/ansible_hosts
$ export ANSIBLE_HOSTS=~/ansible_hosts
```

ANSIBLE\_HOSTS 是 1.9 版本之后开始使用的,代替之前使用的 ANSIBLE\_HOSTS。

可以在 3.1 节找到详细讲解 inventory 文件及使用的内容。

这样 Ansible 系统就安装完成了,后面就可以编写一些脚本开始对 Ansible 进行测试。

## 2. Tar 包安装方式

不想通过 GitHub 提取方式获得 Ansible 的软件包,可以直接下载 Ansible 的 Tar 包,下载地址是 <http://releases.ansible.com/ansible>。这里面存放着从 Ansible 最初发行的 1.1 版本开始,直到最新 1.9.2 版本,到现在共发行过的 42 个软件版本,可以根据需要下载。

Tar 包的安装过程与上述源码安装方式一样,只是源代码获取方式不同而已。

## 3. 制作 rpm 包安装方式

有时需要制作成 rpm 软件包再进行安装。在 GitHub 上 Ansible 项目中提取软件,或直接下载一个 Tar 包,然后使用 make rpm 命令创建 RPM 软件包,最后可分发这个软件包,或者使用它来安装 Ansible。在创建之前,先确定已安装了 rpm-build、make、python2-devel 组件。操作过程如下:

```
$ git clone git://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ~/rpmbuild/ansible-*.noarch.rpm
```

在 Debian/Ubuntu 环境中，可以采用类似的方法制作安装包，制作的命令是 `$ make deb`。

## 2.2.2 用包管理工具安装

### 1. yum 安装方式

要使用 yum 方式安装，需要有合适的 yum 源。Fedora 用户只要连接着因特网，就可以直接使用官方的 yum 源安装。但对于 RHEL、CentOS 的官方 yum 源中没有 Ansible 安装包，这就需要先安装支持第三方的 yum 仓库组件，最常用的有 EPEL、Remi、RPMForge 等。在国内速度较快的高质量 yum 源有中国科技大学 (<http://mirrors.ustc.edu.cn>)、浙江大学 (<http://mirrors.zju.edu.cn/epel/>)、上海交通大学 (<http://ftp.sjtu.edu.cn/fedora/epel/>)、网易 163 (<http://mirrors.163.com>)、sohu 镜像源 (<http://mirrors.sohu.com/fedora-epel/>) 等。

下面安装 EPEL 作为部署 Ansible 的默认 yum 源。

- RHEL (CentOS) 5 版本:

```
rpm -Uvh http://mirrors.zju.edu.cn/epel/6/i386/epel-release-5-4.noarch.rpm
rpm -Uvh http://mirrors.zju.edu.cn/epel/5/x86_64/epel-release-5-4.noarch.rpm
```

- RHEL (CentOS) 6 版本:

```
rpm -Uvh http://mirrors.zju.edu.cn/epel/6/x86_64/epel-release-6-8.noarch.rpm
rpm -Uvh http://mirrors.zju.edu.cn/epel/6/i386/epel-release-6-8.noarch.rpm
```

- RHEL (CentOS) 7 版本:

```
rpm -Uvh http://mirrors.zju.edu.cn/epel/7/x86_64/e/epel-release-7-5.noarch.rpm
```

准备好 yum 源之后，Ansible 就可直接用 yum 命令安装了，命令如下：

```
$ sudo yum install ansible
```

## 2. Apt (Ubuntu) 安装方式

Ubuntu 编译版可在如下地址中获得：<https://launchpad.net/~ansible/+archive/ansible>。

通过执行如下命令直接安装：

```
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

## 3. Homebrew (Mac OSX) 安装方式

在 Mac 系统中，确定已经安装了 Homebrew 之后，直接执行下面命令安装 Ansible：

```
$ brew update
$ brew install Ansible
```

## 4. pip 方式安装

Ansible 也支持可通过 pip 方式安装。pip 是 Python 软件包的安装和管理工具，执行如下命令先安装 pip：

```
$ sudo easy_install pip
```

然后再安装 Ansible：

```
$ sudo pip install ansible
```

如果你是在 OS X 系统上安装，编译器可能会有警告或出错，需要设置 CFLAGS、CPPFLAGS 环境变量：

```
$ sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments pip install
  ansible
```

使用 virtualenv 的读者可通过 virtualenv 安装 Ansible，然而不建议这样做，直接在全局安装 Ansible。不要使用 easy\_install 直接安装 Ansible。

如果把 Ansible 安装在其他相对少见的 Linux 操作系统（如 Gentoo、FreeBSD 等）上，详见 Ansible 官网 <http://docs.ansible.com>。



## 2.3 配置运行环境

### 2.3.1 配置 Ansible 环境

Ansible 配置文件是以 ini 格式存储配置数据的，在 Ansible 中，几乎所有的配置项都可以通过 Ansible 的 playbook 或环境变量来重新赋值。在运行 Ansible 命令时，命令将会按照预先设定的顺序查找配置文件，如下所示：

- 1) `ANSIBLE_CONFIG`：首先，Ansible 命令会检查环境变量，及这个环境变量将指向的配置文件。
- 2) `./ansible.cfg`：其次，将会检查当前目录下的 `ansible.cfg` 配置文件。
- 3) `~/.ansible.cfg`：再次，将会检查当前用户 home 目录下的 `.ansible.cfg` 配置文件。
- 4) `/etc/ansible/ansible.cfg`：最后，将会检查在用软件包管理工具安装 Ansible 时自动产生的配置文件。



**注意** 如果你通过操作系统软件包管理工具或 `pip` 安装，那么你在 `/etc/ansible` 目录下应该已经有了 `ansible.cfg` 配置文件；如果你是通过 GitHub 仓库安装的，在你复制的仓库中 `examples` 目录下可以找到 `ansible.cfg`，你可以把它拷贝到 `/etc/ansible` 目录下。

#### 1. 使用环境变量方式来配置

大多数的 Ansible 参数可以通过设置带有 `ANSIBLE_` 开头的环境变量进行配置，参数名称必须都是大写字母，如下配置项：

```
export ANSIBLE_SUDO_USER=root
```

设置了环境变量之后，`ANSIBLE_SUDO_USER` 就可以在 playbook 中直接引用。

#### 2. 设置 `ansible.cfg` 配置参数

Ansible 有很多配置参数，你也许不会都使用到。下面列出常用的配置参数：

- `inventory`——这个参数表示资源清单 `inventory` 文件的位置，资源清单就是一些 Ansible 需要连接管理的主机列表。在 1.9 版本之前有个类似功能的参数 `hostfile`，但 1.9 版本之后就不建议再使用了。下一章将对资源清单做详细的讲

解。这个参数的配置实例如下：

```
inventory = /etc/ansible/hosts
```

- **library**——Ansible 的操作动作，无论是本地或远程，都使用一小段代码来执行，这小段代码称为模块，这个 **library** 参数就是指向存放 Ansible 模块的目录。配置实例如下：

```
library = /usr/share/ansible
```

Ansible 支持多个目录方式，只要用冒号 (:) 隔开就可以，同时也会检查当前执行 **playbook** 位置下的 **./library** 目录。

- **forks**——设置默认情况下 Ansible 最多能有多少个进程同时工作，默认设置最多 5 个进程并行处理。具体需要设置多少个，可以根据控制主机的性能和被管节点的数量来确定，可能是 50 或 100，默认值 5 是非常保守的设置。配置实例如下：

```
forks = 5
```

- **sudo\_user**——这是设置默认执行命令的用户，也可以在 **playbook** 中重新设置这个参数。配置实例如下：

```
sudo_user = root
```

- **remote\_port**——这是指定连接被管节点的管理端口，默认是 22。除非设置了特殊的 SSH 端口，不然这个参数一般是不需要修改的。配置实例如下：

```
remote_port = 22
```

- **host\_key\_checking**——这是设置是否检查 SSH 主机的密钥。可以设置为 **True** 或 **False**，下一节将详细介绍。配置实例如下：

```
host_key_checking = False
```

- **timeout**——这是设置 SSH 连接的超时间隔，单位是秒。配置实例如下：

```
timeout = 60
```

- **log\_path**——Ansible 系统默认是不记录日志的，如果想把 Ansible 系统的输出记录到日志文件中，需要设置 **log\_path** 来指定一个存储 Ansible 日志的文件。配置实例如下：

```
log_path = /var/log/ansible.log
```

另外需要注意，执行 Ansible 的用户需要有写入日志的权限，模块将会调用被管节点的 **syslog** 来记录，口令是不会出现在日志中的。

### 2.3.2 使用公钥认证

现在运维对安全要求都是比较重视的，一般会采用密钥验证方式来登录，Ansible 1.2.1 之后的版本都默认启用公钥认证。

如果有台被管节点重新安装系统并在 `known_hosts` 中有了与之前不同的密钥信息，就会提示一个密钥不匹配的错误信息，直到被纠正为止。在使用 Ansible 时，如果有台被管节点没有在 `known_hosts` 中被初始化，将会在使用 Ansible 或定时执行 Ansible 时提示对 key 信息的确认。

如果你不想出现这种情况，并且你明白禁用此项行为的含义，只要修改 `home` 目录下 `~/.ansible.cfg` 或 `/etc/ansible/ansible.cfg` 的配置项：

```
[defaults]
host_key_checking = False
```

或者直接在控制主机的操作系统中设置环境变量，如下所示：

```
$export ANSIBLE_HOST_KEY_CHECKING=False
```

需要说明的是，在早期使用 `paramiko` 模式时，公钥认证速度相当慢，因此当使用密钥认证方式时建议采用 SSH 方式连接，这也是现在默认的连接方式。

### 2.3.3 配置 Linux 主机 SSH 无密码访问

为了避免 Ansible 下发指令时输入目标主机密码，通过证书签名达到 SSH 无密码是一个好的方案。推荐使用 `ssh-keygen` 与 `ssh-copy-id` 来实现快速证书的生成及公钥下发，其中 `ssh-keygen` 生产一对密钥，使用 `ssh-copy-id` 来下发生成的公钥。具体操作如下。

在控制主机（`ansiblecontrol`）上创建密钥，执行：`ssh-keygen -t rsa`，有询问直接按回车键即可，将在 `/root/.ssh/` 下生成一对密钥，其中 `id_rsa` 为私钥，`id_rsa.pub` 为公钥，代码如下：

```
[root@ansiblecontrol]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): <回车>
Enter passphrase (empty for no passphrase): <回车>
Enter same passphrase again: <回车>
```

```
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
ec:f2:89:2a:62:c5:1a:53:32:04:04:fd:40:e9:ad:dd root@ansiblecontrol.
ansible.cn
```

The key's randomart image is:

```
+--[ RSA 2048]-----+
|*+..|
|.+|
|..+|
|o..o.|
| =o . S|
|o.o. E.|
| = . .|
|.O. + .|
|.. .... o|
+-----+
```

下发密钥就是控制主机把公钥 `id_rsa.pub` 下发到被管节点上用户下的 `.ssh` 目录，并重命名成 `authorized_keys`，且权限值为 400。接下来推荐常用的密钥拷贝工具 `ssh-copy-id`，把公钥文件 `id_rsa.pub` 公钥拷贝到被管节点，命令格式如下：

```
ssh-copy-id [-h|-?|-n] [-i [identity_file]] [-p port] [[-o <ssh -o
options>] ...] [user@]hostname
```

输入以下命令同步公钥到被管节点 `web1 (192.168.1.111)`：

```
[root@ansiblecontrol]# ssh-copy-id -i /root/.ssh/id_rsa.pub
root@192.168.1.111
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s),
to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you
are prompted now it is to install the new keys
root@192.168.1.111's password:< 输入口令 >
```

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'root@192.168.1.111'"
and check to make sure that only the key(s) you wanted were added.

```
[root@ansiblecontrol]# ssh root@192.168.1.111
```

Last login: Sat Aug 29 12:07:45 2015 from ansiblecontrol.ansible.cn



```
[root@web1 ~]#
```

密钥分发后，需要验证一下 SSH 无密码配置是否成功，只需运行 `ssh root@192.168.1.111`，如果直接进入被管节点 root 账号提示符，即出现 `[root@web1 ~]#`，则说明配置成功。

同样的配置方式对 web2（192.168.1.112）也做下密钥分发。

## 2.4 Ansible 小试身手

Ansible 安装完成之后，我们也了解了基本配置方式，是不是已经急切要动手实践一下 Ansible？下面我们将介绍主机连通性测试和远程执行命令两个小场景，体会一下 Ansible 的便捷、强大。

首先可以先查看一下安装的 Ansible 软件版本信息：

```
$ansible --version
ansible 1.9.2
configured module search path = None
```

### 2.4.1 主机连通性测试

为了使用 Ansible，第一步需要修改主机与组配置，默认的文件在 `/etc/ansible/hosts` 了，格式为 ini，添加两台主机的 IP 地址，同时定义一个 `webservers` 组包含这两个 IP 地址，内容如下：

```
[/etc/Ansible/hosts]
```

```
#web1.ansible.cn
```

```
#web2.ansible.cn
```

```
192.168.1.101
```

```
192.168.1.102
```

```
[webservers]
```

```
#web1.ansible.cn
```

```
#web2.ansible.cn
```

```
192.168.1.101
```

```
192.168.1.102
```


然后，用 `ping` 模块对单台主机进行 `ping` 操作，结果如下：

```
$ ansible 192.168.1.111 -m ping
192.168.1.111 | success >> {
    "changed": false,
    "ping": "pong"
}
```

对 webservers 组进行 ping 操作，结果如下：

```
$ ansible webservers -m ping
192.168.1.111 | success >> {
    "changed": false,
    "ping": "pong"
}
192.168.1.112 | success >> {
    "changed": false,
    "ping": "pong"
}
```

出现上述结果表示 Ansible 正确工作，主机的连通性测试成功，没有对被管节点做任何变更。

 **提示** 这里测试时在控制主机与被管节点之间配置了 SSH 证书信任。如果没有用证书认证，则需要在执行 Ansible 命令时添加 `-k` 参数，在提示“SSH password:”时输入 root（默认）账号密码。实际生产环境中，大多数更倾向于使用 Linux 普通用户账户进行连接并通过 `sudo` 命令实现 root 权限，格式为：

```
ansible webservers -m ping -u ansible -sudo
```

## 2.4.2 在被管节点上批量执行命令

Ansible 很方便进行运维自动化，是运维与研发合作的重要工具。我们借鉴学习开发语言的经典示范程序，也用 Hello World 程序作为自动化运维环境的测试、校验手段。

在用户 home 目录下创建一个资源清单文件 `inventory.cfg`，内容如下：

```
$cat inventory.cfg
[webservers]
192.168.1.101
192.168.1.102
```

用 Ansible 的 shell 模块在 webserver 组的各服务器上显示 “hello ansible!”, 命令如下:

```
$ ansible webserver -m shell -a '/bin/echo hello ansible!' -i
inventory.cfg
192.168.1.111 | success | rc=0 >>
hello ansible!

192.168.1.112 | success | rc=0 >>
hello ansible!
$
```

也可以用 command 模块, 得到类似的结果:

```
$ ansible webserver -m command -a '/bin/echo hello ansible!' -i
inventory.cfg
192.168.1.111 | success | rc=0 >>
hello ansible!

192.168.1.112 | success | rc=0 >>
hello ansible!
```

这样简单的命令就可以完成批量服务器的管理。现在, 你已经来到通往 Ansible 自动化运维的门口, 但是真正巧妙、灵活、功能强大的 Ansible 等待着你深入学习、探索。

## 2.5 获取帮助信息

在 Ansible 1.9.2 版本中有 8 个主要的 Ansible 管理工具, 每个管理工具都是一系列的模块、参数支持。随时可获取的帮助信息对了解掌握 Ansible 系统非常重要。对于 Ansible 每个工具, 都可以简单地在命令后面加上 -h 或 --help 直接获取帮助。

例如, 列出 ansible-doc 工具的支持参数。最主要的参数 -l 列出可使用的模块, -s 列出某个模块支持的动作, 如下所示:

```
[root@ansiblecontrol]# ansible-doc -h
Usage: ansible-doc [options] [module...]

Show Ansible module documentation

Options:
  --version          show program's version number and exit
```

```

-h, --help                show this help message and exit
-M MODULE_PATH, --module-path=MODULE_PATH
                           Ansible modules/ directory
-l, --list                 List available modules
-s, --snippet              Show playbook snippet for specified module(s)
-v                          Show version number and exit

```

用 `ansible-doc -l` 列出 Ansible 系统支持的模块，Ansible 安装后能够列出 259 个模块，如下所示

```

$ ansible-doc -l
less 458 (POSIX regular expressions)
Copyright (C) 1984-2012 Mark Nudelman
less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Homepage: http://www.greenwoodsoftware.com/less
a10_server                Manage A10 Networks AX/SoftAX/Thunder/vThunder...
a10_service_group         Manage A10 Networks AX/SoftAX/Thunder/vThunder...
a10_virtual_server        Manage A10 Networks AX/SoftAX/Thunder/vThunder...
acl                        Sets and retrieves file ACL information.
add_host                   add a host (and alternatively a group) to the ...
airbrake_deployment        Notify airbrake about app deployments
alternatives               Manages alternative programs for common comman...
apache2_module             enables/disables a module of the Apache2 webse...
apt                        Manages apt-packages
apt_key                    Add or remove an apt key
.....

```

`ansible-doc` 直接加模块名称，将显示该模块的描述和使用示例，如 `ansible-doc ping`。每个模块都有一系列的动作，可以用 `ansible-doc -s+` 模块名称列出，如下面列出 `yum` 模块的动作：

```

$ ansible-doc -s yum
less 458 (POSIX regular expressions)
Copyright (C) 1984-2012 Mark Nudelman
less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Homepage: http://www.greenwoodsoftware.com/less
- name: Manages pack age swith the I (yum) pac

```



```

action: yum
  conf_file          # The remote yum configuration file to use
                    # for the
  disable_gpg_check # Whether to disable the GPG checking of
                    # signatures
  disablerepo        # 'Repoid' of repositories to disable for the
                    # insta
  enablerepo         # 'Repoid' of repositories to enable for the
                    # instal
  list               # Various (non-idempotent) commands for usage
                    # with
  name=              # Package name, or package specifier with
                    # version,
  state              # Whether to install ('present', 'latest'),
                    # or remo
  update_cache       # Force updating the cache. Has an effect
                    # only if s

```

另外，在 Ansible 调试自动化脚本时候经常需要获取执行过程的详细信息，可以在命令后面添加 `-v` 或 `-vvv` 得到详细的输出结果。如：

```
$ansible webserver -i inventory.cfg -m ping -vvv
```

最后别忘了经常浏览官方网站 <http://docs.ansible.com/>，其中有详细的使用说明。

## 2.6 本章小结

本章讲解 Ansible 在不同环境下的各种安装方式，以及安装完成之后对系统进行必要的参数配置，对安装后 Ansible 的基本测试，最后介绍如何获取 Ansible 帮助的工具。到这里你应该已经搭建好了基本的 Ansible 测试环境，为后续深入学习 Ansible 提供了必要的基本环境。

下一章将详细介绍 Ansible 涉及的一些基础概念，如何对资源清单进行管理，引入变量、匹配模式等内容。

## Ansible 组件介绍

经过前面 2 章的介绍，我们已经熟悉了 Ansible 的一些安装与简单使用。从本章开始我们将全面介绍 Ansible 的各种组件。这些也是我们使用 Ansible 的过程中必须理解的知识点。本章将通过介绍和讲解 Ansible 日常中经常使用的一些组件，使我们能对 Ansible 有一个全面的了解。

### 3.1 Ansible Inventory

在大规模的配置管理工作中我们需要管理不同业务的不同机器，这些机器的信息都存放在 Ansible 的 Inventory 组件里面。在我们工作中配置部署针对的主机必须先存放在 Inventory 里面，这样才能使用 Ansible 对它进行操作。默认 Ansible 的 Inventory 是一个静态的 INI 格式的文件 `/etc/ansible/hosts`，当然，还可以通过 `ANSIBLE_HOSTS` 环境变量指定或者运行 `ansible` 和 `ansible-playbook` 的时候用 `-i` 参数临时设置。

#### 1. 定义主机和主机组

下面我们来看一下如何在默认的 Inventory 文件中定义一些主机和主机组，具体如下：

```
1 172.17.42.101    ansible_ssh_pass='123456'
2 172.17.42.102    ansible_ssh_pass='123456'
```

```

3 [docker]
4 172.17.42.10[1:3]
5 [docker:vars]
6 ansible_ssh_pass='123456'
7 [ansible:children]
8 docker

```

- 第1行定义了一个主机是 172.17.42.101，然后使用 Inventory 内置变量定义了 SSH 登录密码。
- 第2行定义了一个主机是 172.17.42.102，然后使用 Inventory 内置变量定义了 SSH 登录密码。
- 第3行定义了一个组叫 docker。
- 第4行定义了 docker 组下面 4 台主机从 172.17.42.101 到 172.17.42.103。
- 第5行到第6行针对 docker 组使用 Inventory 内置变量定义了 SSH 登录密码。
- 第7行到第8行定义了一个组叫 ansible，这个组下面包含 docker 组。

ansible\_ssh\_pass 参数是 Ansible Inventory 内置参数，在本节的最后一小节会进行相关的介绍。Inventory 文件一般用来定义远端主机的认证信息，比如 SSH 登录密码、用户名以及 key 相关信息。当然，Inventory 文件也支持主机或者主机组的便利定义。添加完主机和主机组后我们就可以使用 Ansible 命令针对这些主机进行操作和管理了。下面是分别针对不同的主机和主机组进行 Ansible 的 ping 模块检测，ping 模块是 Ansible 中一个连通性检测的模块。当然，Ansible 还内置大量的其他模块，后续章节我们也会慢慢接触。

```

[root@Master ~]# ansible 172.17.42.101:172.17.42.102 -m ping -o
172.17.42.101 | success >> {"changed": false, "ping": "pong"}

```

```

172.17.42.102 | success >> {"changed": false, "ping": "pong"}

```

```

[root@Master ~]# ansible docker -m ping -o
172.17.42.101 | success >> {"changed": false, "ping": "pong"}

```

```

172.17.42.103 | success >> {"changed": false, "ping": "pong"}

```

```

172.17.42.102 | success >> {"changed": false, "ping": "pong"}

```

```

[root@Master ~]# ansible ansible -m ping -o

```

```
172.17.42.102 | success >> {"changed": false, "ping": "pong"}
172.17.42.103 | success >> {"changed": false, "ping": "pong"}
172.17.42.101 | success >> {"changed": false, "ping": "pong"}
```

## 2. 多个 Inventory 列表

通过上一小节对 Inventory 的介绍，我们知道 Ansible 默认的 Inventory 文件是一个 INI 的静态文件，其实 Ansible 还支持多个 Inventory 文件，这样我们就可以很方便地管理不同业务或者不同环境中的机器了。如何使用多个 Inventory 文件呢？

首先需要修改 `ansible.cfg` 中 `hosts` 文件的定义，或者使用 `ANSIBLE_HOSTS` 环境变量定义。这里我们准备一个文件夹，里面将存放多个 Inventory 文件，如以下目录所示：

```
[root@Master ~]# tree inventory/
inventory/
├── docker
└── hosts
```

不同的文件可以存放不同的主机，我们来分别看一下文件的内容：

```
[root@Master ~]# cat inventory/hosts
172.17.42.101 ansible_ssh_pass='123456'
172.17.42.102 ansible_ssh_pass='123456'
[root@Master ~]# cat inventory/docker
[docker]
172.17.42.10[1:3]
[docker:vars]
ansible_ssh_pass='123456'
[ansible:children]
docker
```

最后我们修改了 `ansible.cfg` 文件中 `inventory` 的值，这里不再指向一个文件，而是指向一个目录，修改如下：

```
inventory = /root/inventory/
```

这样我们就可以使用 Ansible 的 `list-hosts` 参数来进行如下验证：

```
[root@Master ~]# ansible 172.17.42.101:172.17.42.102 --list-hosts
```



```

172.17.42.101
172.17.42.102
[root@Master ~]# ansible docker --list-hosts
172.17.42.101
172.17.42.102
172.17.42.103
[root@Master ~]# ansible ansible --list-hosts
172.17.42.101
172.17.42.102
172.17.42.103

```

其实 Ansible 中的多个 Inventory 跟单个文件没什么区别，我们也可以容易定义或者引用多个 Inventory，甚至可以把不同环境的主机或者不同业务的主机放在不同的 Inventory 文件里面，方便日后维护。

### 3. 动态 Inventory

在实际应用部署中会有大量的主机列表。如果手动维护这些列表将是一个非常繁琐的事情。其实 Ansible 还支持动态的 Inventory，动态 Inventory 就是 Ansible 所有的 Inventory 文件里面的主机列表和变量信息都支持从外部拉取。比如我们可以从 CMDB 系统和 Zabbix 监控系统拉取所有的主机信息，然后使用 Ansible 进行管理。这样一来我们就可以很方便地将 Ansible 与其他运维系统结合起来。关于引用动态 Inventory 的功能配置起来也很简单。我们只需要把 ansible.cfg 文件中 inventory 的定义值改成一个执行脚本即可。这个脚本的内容不受任何编程语言限制，但是这个脚本使用参数时有一定的规范并且对脚本执行的结果也有要求。这个脚本需要支持两个参数：

- --list 或者 -l，这个参数运行后会显示所有的主机以及主机组的信息（JSON 格式）。
- --host 或者 -H，这个参数后面需要指定一个 host，运行结果会返回这台主机的所有信息（包括认证信息、主机变量等），也是 JSON 格式。

下面我们通过一个简单的例子了解动态 Inventory 实现流程。这里编写了一个简单 hosts.py 脚本，代码如下：

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
import argparse
import sys

```

```

import json

def lists():
    r = {}
    h=[ '172.17.42.10' + str(i) for i in range(1,4) ]
    hosts={'hosts': h}
    r['docker'] = hosts
    return json.dumps(r,indent=4)

def hosts(name):
    r = {'ansible_ssh_pass': '123456'}
    cpis=dict(r.items())
    return json.dumps(cpis)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-l', '--list', help='hosts list', action='store_true')
    parser.add_argument('-H', '--host', help='hosts vars')
    args = vars(parser.parse_args())

    if args['list']:
        print lists()
    elif args['host']:
        print hosts(args['host'])
    else:
        parser.print_help()

```

这个脚本定义了两个函数：lists 函数在指定 --list 参数后执行，hosts 函数会在指定 --host 参数后执行。脚本的每个函数都会返回一个 JSON，运行结果如下：

```

[root@Master ~]# python hosts.py --list
{
  "docker": {
    "hosts": [
      "172.17.42.101",
      "172.17.42.102",
      "172.17.42.103"
    ]
  }
}

[root@Master ~]# python hosts.py -H 172.17.42.102
{"ansible_ssh_pass": "123456"}

```

这里定义了一个 docker 组且组里定义了 3 台主机，然后定义每台设备的 SSH 密码。ansible\_ssh\_pass 是 Inventory 内置的参数，下一节会详细解释。最后我们来执行临时指定这个 hosts.py 脚本，没修改 ansible.cfg 文件，运行结果如下：

```
[root@Master ~]# ansible -i hosts.py 172.17.42.102:172.17.42.103 -m ping -o
172.17.42.102 | success >> {"changed": false, "ping": "pong"}

172.17.42.103 | success >> {"changed": false, "ping": "pong"}

[root@Master ~]# ansible -i hosts.py docker -m ping -o
172.17.42.101 | success >> {"changed": false, "ping": "pong"}

172.17.42.102 | success >> {"changed": false, "ping": "pong"}

172.17.42.103 | success >> {"changed": false, "ping": "pong"}
```

本节只是简单地编写了一个 Inventory 脚本，希望读者能理解这个原理，在实际工作中可以根据自己的需求编写相应的脚本。目前官方也有一些关于从 cobbler 和 AWS 获取主机或者主机组相关的 Inventory 脚本，有兴趣的读者可以自己查阅。

4. Inventory 内置参数

这里介绍 Ansible Inventory 内置的一些参数，这些参数在我们实际工作中也会经常使用，我们可以直接在 Inventory 文件中定义它，当然动态的 Inventory 也可以使用它，如表 3-1 所示。

表 3-1 Inventory 内置参数

参数	解释	例子
ansible_ssh_host	定义 host ssh 地址	ansible_ssh_host=192.168.1.117
ansible_ssh_port	定义 hosts ssh 端口	ansible_ssh_port=5000
ansible_ssh_user	定义 hosts ssh 认证用户	ansible_ssh_user=yadmin
ansible_ssh_pass	定义 hosts ssh 认证密码	ansible_ssh_user='123456'
ansible_sudo	定义 hosts sudo 用户	ansible_sudo=yadmin
ansible_sudo_pass	定义 hosts sudo 密码	ansible_sudo_pass='123456'
ansible_sudo_exe	定义 hosts sudo 路径	ansible_sudo_exe=/usr/bin/sudo
ansible_connection	定义 hosts 连接方式	ansible_connection=local
ansible_ssh_private_key_file	定义 hosts 私钥	ansible_ssh_private_key_file=/root/key
ansible_shell_type	定义 hosts shell 类型	ansible_shell_type=zsh

(续)

参数	解释	例子
<code>ansible_python_interpreter</code>	定义 hosts 任务执行 python 路径	<code>ansible_python_interpreter=/usr/bin/python2.6</code>
<code>ansible_*_interpreter</code>	定义 hosts 其他语言解析器路径	<code>ansible_ruby_interpreter=/usr/bin/ruby</code>

## 3.2 Ansible Ad-Hoc 命令

我们经常会通过命令行的形式使用 Ansible 模块，Ansible 自带很多模块，可以直接使用这些模块。目前 Ansible 已经自带了 259 个模块，我们可以使用 `ansible-doc -l` 显示所有自带模块，还可以通过 `ansible-doc “模块名”`，查看模块的介绍以及案例。需要注意的是，如果使用 `Ah-hoc` 命令，Ansible 的一些插件功能就无法使用，比如 `loop facts` 功能等。本节就介绍一些日常的 Ad-Hoc 命令。

### 1. 执行命令

Ansible 命令都是并发执行的，我们可以针对目标主机执行任何命令。默认的并发数目由 `ansible.cfg` 中的 `forks` 值来控制。当然，也可以在运行 Ansible 命令的时候通过 `-f` 指定并发数。如果碰到执行任务时间很长的情况，也可以使用 Ansible 的异步执行功能来执行。下面我们简单地测试一下：

```
[root@Master ~]# ansible docker -m shell -a 'hostname' -o
172.17.42.101 | success | rc=0 | (stdout) 4b461620612a
172.17.42.103 | success | rc=0 | (stdout) 703bb6924049
172.17.42.102 | success | rc=0 | (stdout) 24e575b23394

[root@Master ~]# ansible docker -m shell -a 'uname -r' -f 5 -o
172.17.42.103 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
172.17.42.102 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
172.17.42.101 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
```

使用异步执行功能，`-P 0` 的情况下会直接返回 `job_id`，然后针对主机根据 `job_id` 查询执行结果：

```
[root@Master ~]# ansible docker -B 120 -P 0 -m shell -a 'sleep
10;hostname' -f 5 -o
background launch...
```



```
172.17.42.103 | success >> {"ansible_job_id": "288872560652.1072", "results_
file": "/root/.ansible_async/288872560652.1072", "started": 1}
```

```
172.17.42.101 | success >> {"ansible_job_id": "288872560652.1096", "results_
file": "/root/.ansible_async/288872560652.1096", "started": 1}
```

```
172.17.42.102 | success >> {"ansible_job_id": "288872560652.1097", "results_
file": "/root/.ansible_async/288872560652.1097", "started": 1}
```

每台主机会产生不同的 job\_id，可以通过 async\_status 模块查看异步任务的状态和结果：

```
[root@Master ~]# ansible 172.17.42.101 -m async_status -a
'jid=288872560652.1096'
```

```
172.17.42.101 | success >> {
  "ansible_job_id": "288872560652.1096",
  "changed": true,
  "cmd": "sleep 10;hostname",
  "delta": "0:00:10.010299",
  "end": "2015-06-07 12:01:02.553748",
  "finished": 1,
  "rc": 0,
  "start": "2015-06-07 12:00:52.543449",
  "stderr": "",
  "stdout": "4b461620612a",
  "warnings": []
}
```

当 -P 参数大于 0 的时候，Ansible 会自动根据 job\_id 去轮询查询执行结果：

```
[root@Master ~]# ansible docker -B 12 -P 1 -m shell -a 'sleep
5;hostname' -f 5 -o
background launch...
```

```
172.17.42.103 | success >> {"ansible_job_id": "9195868444.1102", "results_
file": "/root/.ansible_async/9195868444.1102", "started": 1}
```

```
172.17.42.101 | success >> {"ansible_job_id": "9195868444.1150", "results_
file": "/root/.ansible_async/9195868444.1150", "started": 1}
```

```
172.17.42.102 | success >> {"ansible_job_id": "9195868444.1127", "results_
file": "/root/.ansible_async/9195868444.1127", "started": 1}
```

```

172.17.42.102 | success >> {"ansible_job_id": "9195868444.1127",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1127", "started": 1}

172.17.42.103 | success >> {"ansible_job_id": "9195868444.1102",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1102", "started": 1}

172.17.42.101 | success >> {"ansible_job_id": "9195868444.1150",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1150", "started": 1}

<job 9195868444.1127> polling on 172.17.42.102, 11s remaining
<job 9195868444.1102> polling on 172.17.42.103, 11s remaining
<job 9195868444.1150> polling on 172.17.42.101, 11s remaining
172.17.42.101 | success >> {"ansible_job_id": "9195868444.1150",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1150", "started": 1}

172.17.42.102 | success >> {"ansible_job_id": "9195868444.1127",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1127", "started": 1}

172.17.42.103 | success >> {"ansible_job_id": "9195868444.1102",
    "changed": false, "finished": 0, "results_file": "/root/.ansible_
    async/9195868444.1102", "started": 1}

<job 9195868444.1127> polling on 172.17.42.102, 10s remaining
<job 9195868444.1102> polling on 172.17.42.103, 10s remaining
<job 9195868444.1150> polling on 172.17.42.101, 10s remaining
172.17.42.101 | success >> {"ansible_job_id": "9195868444.1150",
    "changed": true, "cmd": "sleep 5;hostname", "delta": "0:00:05.008892",
    "end": "2015-06-07 12:04:04.221857", "finished": 1, "rc": 0, "start":
    "2015-06-07 12:03:59.212965", "stderr": "", "stdout": "4b461620612a",
    "warnings": []}

172.17.42.102 | success >> {"ansible_job_id": "9195868444.1127",
    "changed": true, "cmd": "sleep 5;hostname", "delta": "0:00:05.016063",
    "end": "2015-06-07 12:04:04.156382", "finished": 1, "rc": 0, "start":
    "2015-06-07 12:03:59.140319", "stderr": "", "stdout": "24e575b23394",
    "warnings": []}

172.17.42.103 | success >> {"ansible_job_id": "9195868444.1102",

```

```

"changed": true, "cmd": "sleep 5;hostname", "delta": "0:00:05.017769",
"end": "2015-06-07 12:04:04.151042", "finished": 1, "rc": 0, "start":
"2015-06-07 12:03:59.133273", "stderr": "", "stdout": "703bb6924049",
"warnings": []}

<job 9195868444.1127> finished on 172.17.42.102 => {
  "ansible_job_id": "9195868444.1127",
  "changed": true,
  "cmd": "sleep 5;hostname",
  "delta": "0:00:05.016063",
  "end": "2015-06-07 12:04:04.156382",
  "finished": 1,
  "invocation": {
    "module_args": "jid=9195868444.1127",
    "module_name": "async_status"
  },
  "rc": 0,
  "start": "2015-06-07 12:03:59.140319",
  "stderr": "",
  "stdout": "24e575b23394",
  "warnings": []
}

<job 9195868444.1102> finished on 172.17.42.103 => {
  "ansible_job_id": "9195868444.1102",
  "changed": true,
  "cmd": "sleep 5;hostname",
  "delta": "0:00:05.017769",
  "end": "2015-06-07 12:04:04.151042",
  "finished": 1,
  "invocation": {
    "module_args": "jid=9195868444.1102",
    "module_name": "async_status"
  },
  "rc": 0,
  "start": "2015-06-07 12:03:59.133273",
  "stderr": "",
  "stdout": "703bb6924049",
  "warnings": []
}

<job 9195868444.1150> finished on 172.17.42.101 => {
  "ansible_job_id": "9195868444.1150",
  "changed": true,
  "cmd": "sleep 5;hostname",
  "delta": "0:00:05.008892",

```

```

"end": "2015-06-07 12:04:04.221857",
"finished": 1,
"invocation": {
  "module_args": "jid=9195868444.1150",
  "module_name": "async_status"
},
"rc": 0,
"start": "2015-06-07 12:03:59.212965",
"stderr": "",
"stdout": "4b461620612a",
"warnings": []
}

```

## 2. 复制文件

我们还可以使用 `copy` 模块来批量下发文件，文件的变化是通过 MD5 值来判断的，如下所示：

```

[root@Master ~]# ansible docker -m copy -a 'src=hosts.py dest=/root/
hosts.py owner=root group=root mode=644 backup=yes' -o

```

```

172.17.42.103 | success >> {"changed": true, "checksum": "01800be332e
6f2ff276e5a5e9c2a33c939c0388a", "dest": "/root/hosts.py", "gid":
0, "group": "root", "md5sum": "7434e7fe32815562fd18e2e10457bd40",
"mode": "0644", "owner": "root", "size": 736, "src": "/root/.
ansible/tmp/ansible-tmp-1433677195.35-100509674775547/source",
"state": "file", "uid": 0}

```

```

172.17.42.101 | success >> {"changed": true, "checksum": "01800be332e
6f2ff276e5a5e9c2a33c939c0388a", "dest": "/root/hosts.py", "gid":
0, "group": "root", "md5sum": "7434e7fe32815562fd18e2e10457bd40",
"mode": "0644", "owner": "root", "size": 736, "src": "/root/.
ansible/tmp/ansible-tmp-1433677195.43-169279680329648/source",
"state": "file", "uid": 0}

```

```

172.17.42.102 | success >> {"changed": true, "checksum": "01800be
332e6f2ff276e5a5e9c2a33c939c0388a", "dest": "/root/hosts.py",
"gid": 0, "group": "root", "md5sum": "7434e7fe32815562fd18e2e104
57bd40", "mode": "0644", "owner": "root", "size": 736, "src": "/
root/.ansible/tmp/ansible-tmp-1433677195.36-58360851137130/source",
"state": "file", "uid": 0}

```

我们再来验证文件下发功能，如下所示：



```
[root@Master ~]# ansible docker -m shell -a 'md5sum /root/hosts.py' -f
5 -o
172.17.42.101 | success | rc=0 | (stdout) 7434e7fe32815562fd18e2e10457
bd40 /root/hosts.py
172.17.42.102 | success | rc=0 | (stdout) 7434e7fe32815562fd18e2e10457
bd40 /root/hosts.py
172.17.42.103 | success | rc=0 | (stdout) 7434e7fe32815562fd18e2e10457
bd40 /root/hosts.py
```

### 3. 包和服务管理

Ansible 还支持直接使用 Ad-Hoc 命令来管理包和服务，如下所示：

```
[root@Master ~] ansible docker -m yum -a 'name=httpd state=latest' -f 5 -o

[root@Master ~]# ansible docker -m service -a 'name=httpd state=started
' -f 5 -o
172.17.42.101 | success >> {"changed": false, "name": "httpd", "state":
"started"}
172.17.42.103 | success >> {"changed": false, "name": "httpd", "state":
"started"}
172.17.42.102 | success >> {"changed": false, "name": "httpd", "state":
"started"}

[root@Master ~]# ansible docker -m shell -a 'rpm -qa httpd' -f 5 -o
172.17.42.101 | success | rc=0 | (stdout) httpd-2.2.15-39.el6.centos.
x86_64
172.17.42.102 | success | rc=0 | (stdout) httpd-2.2.15-39.el6.centos.
x86_64
172.17.42.103 | success | rc=0 | (stdout) httpd-2.2.15-39.el6.centos.
x86_64
```

验证服务运行情况：

```
[root@Master ~]# ansible docker -m shell -a 'netstat -tln|grep httpd'
-f 5
172.17.42.102 | success | rc=0 >>
tcp      0      0 :::80          :::*
LISTEN   799/httpd

172.17.42.101 | success | rc=0 >>
tcp      0      0 :::80          :::*
```

```
LISTEN      798/httpd
```

```
172.17.42.103 | success | rc=0 >>
```

```
tcp        0      0 :::80          :::*
```

```
LISTEN      752/httpd
```

#### 4. 用户管理

首先通过 `openssl` 命令来生成一个密码，因为 `ansible user` 的 `password` 参数需要接受加密后的值，如下所示：

```
[root@Master ~]# echo ansible | openssl passwd -1 -stdin
$1$GPMku7yL$.qu3NC2geUv0J.NvgfCio1
```

然后使用 `user` 模块批量新建用户：

```
[root@Master ~]# ansible docker -m user -a 'name=shencan
password="$1$GPMku7yL$.qu3NC2geUv0J.NvgfCio1" -f 5 -o
```

```
172.17.42.103 | success >> {"changed": true, "comment": "",
"createhome": true, "group": 500, "home": "/home/shencan", "name":
"shencan", "password": "NOT_LOGGING_PASSWORD", "shell": "/bin/bash",
"state": "present", "system": false, "uid": 500}
```

```
172.17.42.102 | success >> {"changed": true, "comment": "",
"createhome": true, "group": 500, "home": "/home/shencan", "name":
"shencan", "password": "NOT_LOGGING_PASSWORD", "shell": "/bin/bash",
"state": "present", "system": false, "uid": 500}
```

```
172.17.42.101 | success >> {"changed": true, "comment": "",
"createhome": true, "group": 500, "home": "/home/shencan", "name":
"shencan", "password": "NOT_LOGGING_PASSWORD", "shell": "/bin/bash",
"state": "present", "system": false, "uid": 500}
```

最后我们通过 SSH 登录来验证前面新建用户是否成功，密码就是前面的 `ansible`。

```
[root@Master ~]# ssh 172.17.42.103 -l shencan
```

```
shencan@172.17.42.103's password:
```

```
[shencan@703bb6924049 ~]$ logout
```

```
Connection to 172.17.42.103 closed.
```

关于 Ansible 的其他 Ad-Hoc 模块或者模块用法，可以通过 `ansible-doc -l` 和 `ansible-doc 模块名` 进行查看。

### 3.3 Ansible playbook

playbook 是 Ansible 进行配置管理的组件，虽然 Ansible 的日常 Ad-Hoc 命令功能很强大，能完成一些基本配置管理工作，但是 Ad-Hoc 命令无法支撑复杂环境的配置管理工作。在我们实际使用 Ansible 的工作中，大部分时间都是在编写 playbook，这是 Ansible 非常重要的组件之一，所以第 4 章将用一整章的篇幅来详细讲解 Ansible 的 playbook。

### 3.4 Ansible facts

facts 组件是 Ansible 用于采集被管机器设备信息的一个功能，我们可以使用 setup 模块查机器的所有 facts 信息，可以使用 filter 来查看指定信息。整个 facts 信息被包装在一个 JSON 格式的数据结构中，ansible\_facts 是最上层的值。下面我们通过实际操作来简单了解 facts 的数据结构：

```
[root@Master ~]# ansible 172.17.42.101 -m setup
```

```
172.17.42.101 | success >> {
```

```
  "ansible_facts": {
```

```
    "ansible_all_ipv4_addresses": [
```

```
      "172.17.0.2",
```

```
      "172.17.42.101"
```

```
    ],
```

```
    "ansible_all_ipv6_addresses": [
```

```
      "fe80::42:acff:fe11:2",
```

```
      "fe80::9093:d8ff:fe7d:f6d4"
```

```
    ],
```

```
    "ansible_architecture": "x86_64",
```

```
    "ansible_bios_date": "12/01/2006",
```

```
    "ansible_bios_version": "VirtualBox",
```

```
    "ansible_cmdline": {
```

```
      "KEYBOARDTYPE": "pc",
```

```
      "KEYTABLE": "us",
```

```
      "LANG": "zh_CN.UTF-8",
```

```
      "crashkernel": "auto",
```

```
      "quiet": true,
```

```
      "rd_LVM_LV": "vg_master/lv_root",
```

```
      "rd_NO_DM": true,
```

```
"rd_NO_LUKS": true,
"rd_NO_MD": true,
"rhgb": true,
"ro": true,
"root": "/dev/mapper/vg_master-lv_root"
```

```
},
```

----- 此处省略 N 行 -----

所有的数据格式都是 JSON 格式，facts 还支持查看指定信息，比如下面只查看设备的 ipv4 地址：

```
[root@Master ~]# ansible 172.17.42.101 -m setup -a 'filter=ansible_
all_ipv4_addresses'
172.17.42.101 | success >> {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "172.17.0.2",
      "172.17.42.101"
    ]
  },
  "changed": false
}
```

facts 组件默认已经收集了很多的设备基础信息，这些信息可以在做配置管理的时候进行引用。下一章也会介绍把 facts 信息直接当作 playbook 变量信息进行引用。后面章节也会介绍如何扩展 facts 进行信息收集，可以定制 facts 以便收集我们想要的信息。下面介绍通过 facter 和 ohai 来扩展 facts 信息。

### 1. 使用 facter 扩展 facts 信息

使用过 Puppet 的读者都熟悉 facter 是 Puppet 里面一个负责收集主机静态信息的组件，Ansible 的 facts 功能也一样。Ansible 的 facts 组件也会判断被控制机器上是否安装有 facter 和 ruby-json 包，如果存在的话，Ansible 的 facts 也会采集 facter 信息。我们来看以下机器信息：

```
[root@Master ~]# ansible 172.17.42.101 -m shell -a 'rpm -qa ruby-json
facter'
172.17.42.101 | success | rc=0 >>
facter-1.6.18-8.el6.x86_64
ruby-json-1.4.6-1.el6.x86_64
```



然后运行 `facter` 模块查看 `facter` 信息：

```
[root@Master ~]# ansible 172.17.42.101 -m facter
```

```
172.17.42.101 | success >> {
  "architecture": "x86_64",
  "changed": false,
  "facterversion": "1.6.18",
  "hardwareisa": "x86_64",
  "hardwaremodel": "x86_64",
  "hostname": "4b461620612a",
  "id": "root",
  "interfaces": "eth0,eth1,lo",
  "ipaddress": "172.17.0.2",
  "ipaddress_eth0": "172.17.0.2",
  "ipaddress_eth1": "172.17.42.101",
  "ipaddress_lo": "127.0.0.1",
  "is_virtual": "true",
  "kernel": "Linux",
  "kernelmajversion": "3.10",
  "kernelrelease": "3.10.5-3.el6.x86_64",
```

----- 省略 N 行 -----

当然，如果直接运行 `setup` 模块也会采集 `facter` 信息，如下所示：

```
[root@Master ~]# ansible 172.17.42.101 -m setup
```

```
172.17.42.101 | success >> {
```

----- 省略 N 行 -----

```
  "facter_swapfree": "991.48 MB",
  "facter_swapspace": "1024.00 MB",
  "facter_timezone": "UTC",
  "facter_uniqueid": "11ac0200",
  "facter_uptime": "8:41 hours",
  "facter_uptime_days": 0,
  "facter_uptime_hours": 8,
  "facter_uptime_seconds": 31299,
  "facter_virtual": "virtualbox",
  "module_setup": true
```

```
},
```

```
"changed": false
```

```
}
```

所有 `facter` 信息在 `ansible_facts` 下以 `facter_` 开头，这些信息的引用方式跟 Ansible 自带 `facts` 组件收集的信息引用方式一致。

## 2. 使用 ohai 扩展 facts 信息

`ohai` 是 Chef 配置管理工具中检测节点属性的工具，Ansible 的 `facts` 也支持 `ohai` 信息的采集。当然需要被管机器上安装 `ohai` 包。下面介绍 `ohai` 相关信息的采集：

```
[root@Master ~]# ansible 172.17.42.1 -m shell -a 'gem list|grep ohai'
172.17.42.1 | success | rc=0 >>
ohai (8.4.0)
```

如果主机上没有安装 `ohai` 包，可以使用 `gem` 方式进行安装。如果存在 `ohai` 包，可以直接运行 `ohai` 模块查看 `ohai` 属性：

```
[root@Master ~]# ansible 172.17.42.1 -m ohai
172.17.42.1 | success >> {
----- 省略 N 行 -----
    "ohai_time": 1433689885.3133919,
    "os": "linux",
    "os_version": "3.10.5-3.el6.x86_64",
    "platform": "centos",
    "platform_family": "rhel",
    "platform_version": "6.5",
    "root_group": "root",
    "uptime": "9 hours 00 minutes 06 seconds",
    "uptime_seconds": 32406,
    "virtualization": {
        "role": "guest",
        "system": "vbox",
        "systems": {
            "vbox": "guest"
        }
    }
}
```

如果直接运行 `setup` 模块，也会采集 `ohai` 信息：

```
[root@Master ~]# ansible 172.17.42.1 -m setup
172.17.42.1 | success >> {
    "ansible_facts": {
```

```

----- 此处省略 N 行 -----
"ohai_ohai_time": 1433690048.8647399,
"ohai_os": "linux",
"ohai_os_version": "3.10.5-3.el6.x86_64",
"ohai_platform": "centos",
"ohai_platform_family": "rhel",
"ohai_platform_version": "6.5",
"ohai_root_group": "root",
"ohai_uptime": "9 hours 02 minutes 50 seconds",
"ohai_uptime_seconds": 32570,
"ohai_virtualization": {
    "role": "guest",
    "system": "vbox",
    "systems": {
        "vbox": "guest"
    }
},
"changed": false
}

```

所有的 ohai 信息在 `ansible_facts` 下以 `ohai_` 开头，这些信息的引用方式跟 Ansible 自带 facts 组件收集的信息引用方式一致。

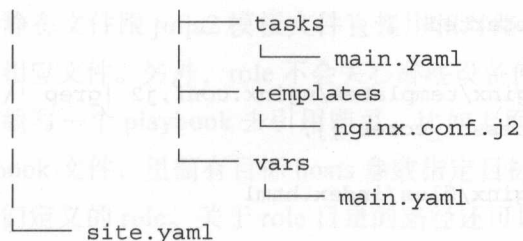
### 3.5 Ansible role

Ansible 在 1.2 版本以后就支持了 role。在实际工作中有很多不同业务需要编写很多 playbook 文件，如果时间一久，对这些 playbook 文件很难进行维护，这个时候我们就可以采用 role 的方式管理 playbook。其实 role 只是对我们日常使用的 playbook 的目录结构进行一些规范，与日常的 playbook 没什么区别。下面通过一个案例来介绍 role 相关的目录规范，如下所示：

```

|— roles
|   |— nginx
|       |— files
|           |— index.html
|       |— handlers
|       |— main.yaml

```



7 directories, 6 files

这里简单了定义了一个 role，它的主要工作就是配置部署 Nginx 服务。role 的所有文件内容都在 nginx 目录下。跟 role 同级别的还有一个 site.yaml 文件，这个文件就是 role 引用的入口文件，文件的名称可以随意定义。files 目录里面存放一些静态文件，handlers 目录里面存放一些 task 的 handler。tasks 目录里面就是我们平常写的 playbook 中的 task。templates 目录里面存放着 jinja2 模板文件。vars 目录下存放着变量文件。下面分别来查看每个文件的内容：

```
[root@Master nginx]# cat site.yaml
```

```
---
- hosts: 172.17.42.103
  roles:
    - { role: nginx, version: 1.0.15 }
```

```
[root@Master nginx]# cat roles/nginx/tasks/main.yaml
```

```
---
- name: Install nginx package
  yum: name=nginx-{{ version }} state=present
- name: Copy nginx.conf Template
  template: src=nginx.conf.j2 dest=/etc/nginx/nginx.conf owner=root
            group=root backup=yes mode=0644
  notify: restart nginx
- name: Copy index.html
  copy: src=index.html dest=/usr/share/nginx/html/index.html
        owner=root group=root backup=yes mode=0644
- name: make sure nginx service running
  service: name=nginx state=started
```

```
[root@Master nginx]# cat roles/nginx/handlers/main.yaml
```

```
---
- name: restart nginx
```



```

service: name=nginx state=restarted

[root@Master nginx]# cat roles/nginx/templates/nginx.conf.j2 |grep '\{\{'
worker_processes  {{ ansible_processor_cores }};

[root@Master nginx]# cat roles/nginx/files/index.html
hello kugou

```

上面介绍了 Nginx role 的所有文件。如果以后需要对 role 进行修改或者调整，只需修改相应的文件即可。如果还想把这个 Nginx role 分享给其他朋友，也只需把整个目录分享即可。下面执行这个 role：

```

[root@Master nginx]# ansible-playbook -i /root/hosts site.yaml

PLAY [172.17.42.103] *****
GATHERING FACTS *****
ok: [172.17.42.103]

TASK: [nginx | Install nginx package] *****
changed: [172.17.42.103]

TASK: [nginx | Copy nginx.conf Template] *****
changed: [172.17.42.103]

TASK: [nginx | Copy index.html] *****
changed: [172.17.42.103]

TASK: [nginx | make sure nginx service running] *****
changed: [172.17.42.103]

NOTIFIED: [nginx | restart nginx] *****
changed: [172.17.42.103]

PLAY RECAP *****
172.17.42.103      : ok=6    changed=5    unreachable=0    failed=0

```

role 执行完成后，通过 curl 命令进行 Nginx 服务测试。

```

[root@Master nginx]# curl 172.17.42.103
hello kugou

```

role 文件的内容还是比较简单的，如果我们定义一个 role，然后在写 playbook 的

时候静态文件跟 jinja2 模板文件直接用相对路径就行，Ansible 会自动去相应的目录下寻找相应文件。另外，role 不会关心哪些设备使用它，它只是关于一个功能的集合，只需要编写一个 playbook 去引用即可。比如上面的 site.yaml 文件，它就是一个简单的 playbook 文件，里面有目标 hosts 参数指定目标主机，然后它会引用一个 role 参数去调用我们定义的 role。关于 role 目录的路径还可以使用 ansible.cfg 中的 roles\_path 进行指定，也可以引用当前目录下的 roles 目录。

## 3.6 Ansible Galaxy

Ansible 的 Galaxy 是 Ansible 官方一个分享 role 的功能平台，它的网址是 <https://galaxy.ansible.com/list#/roles>。可以把你编写的 role 通过 ansible-galaxy 上传到 Galaxy 网站供其他人下载和使用。可以通过 ansible-galaxy 命令很简单地实现 role 的分享和安装。当然 Ansible 也支持直接从 GitHub 上下载 role。在我们使用 ansible-galaxy 命令下载 role 的时候需要了解 role 的运行平台和 Ansible 依赖版本以及相关依赖，等等。日常工作中我们使用 ansible-galaxy install 就可以，默认会安装到 /etc/ansible/roles/ 目录下，其引用跟我们自己写的 role 引用方式一样。

## 3.7 本章小结

通过本章的学习我们已经了解 Ansible 一些常用的组件，并对 Ansible Ad-Hoc 命令中常用的几个命令进行了演示，我们还介绍了 Ansible playbook、facts、role、Galaxy 等。学习一个软件，首先得了解这个软件的一些常用组件，以及如何使用这个软件。下一章将介绍 Ansible 在做配置管理工作中最重要的一个组件 playbook。

## playbook 详解

通过第3章的学习我们已经对 Ansible 的一些组件有了一定了解，也知道了每个组件的一些应用场景，这一章我们将讲解 Ansible 最核心的组件 `playbook`。大家都知道我们使用 Ansible 很大一部分工作都是进行配置管理工作。在实际工作中我们也会去大量编写和使用 `playbook`，作为 Ansible 最核心的功能组件，本章也会通过大量的篇幅去详细讲解 `playbook`。首先我们对 `playbook` 的基础语法进行了解，包括一些常用的 `ansible-playbook` 命令参数，然后会详细介绍 Ansible 的变量定义以及如何在 `playbook` 内引用变量，接着我们会介绍在 `playbook` 里面使用 `loops` `lookups` `conditionals`，最后我们还会介绍关于 `jinja2` 模板的一些常用 `filter`。

### 4.1 `playbook` 基本语法

这一节介绍 `playbook` 的基本语法以及常用命令。

Ansible 的 `playbook` 文件格式为 `YAML` 语法，所以希望读者在编写 `playbook` 之前对 `YAML` 语法有一定的了解，否则在运行 `playbook` 的时候会经常碰到语法错误的提示。这里只介绍 `nginx.yaml` 这个 `playbook`，关于 `YAML` 语法的介绍可以通过 <http://www.yaml.org/spec/1.2/spec.html#Syntax> 网站进行学习与了解。

下面我们通过一个安装部署 Nginx 服务的案例开始，首先我们来看这个 playbook 代码：

```
[root@python ~]# cat nginx.yaml
1  ---
2  - hosts: all
3    tasks:
4      - name: Install Nginx Package
5        yum: name=nginx state=present
6
7      - name: Copy Nginx.conf
8        template: src=./nginx.conf.j2 dest=/etc/nginx/nginx.conf
9                  owner=root group=root mode=0644 validate='nginx -t -c %s'
10       notify:
11         - ReStart Nginx Service
12
13     handlers:
14       - name: ReStart Nginx Service
15         service: name=nginx state=restarted
```

- 第1行表示该文件是 YAML 文件，非必须。
- 第2行定义该 playbook 针对的目标主机，all 表示针对所有主机，这个参数支持 Ansible Ad-Hoc 模式的所有参数。
- 第3行定义该 playbook 所有的 tasks 集合，比如下面我们定义的 3 个 task。
- 第4行定义一个 task 的名称，非必须，建议根据 task 实际任务命名。
- 第5行定义一个状态的 action，比如这里使用 yum 模块实现 Nginx 软件包的安装。
- 第6行到第9行使用 template 模板去管理 /etc/nginx/nginx.conf 文件，owner group 定义该文件的属主以及属组，使用 validate 参数指文件生成后使用 nginx -t -c %s 命令去做 Nginx 文件语法验证，notify 是触发 handlers，如果同步后，文件的 MD5 值有变化会触发 ReStart Nginx Service 这个 handler。
- 第10行到第12行是定义一个 handler 状态让 Nginx 服务重启，handler 的名称是 ReStart Nginx Service。

接下来我们来看 Inventory 的 hosts 主机文件以及 nginx.conf.j2 模板文件的内容：

```
[root@python ~]# cat hosts
```



```
[nginx]
192.168.1.11[6:8]
[nginx:vars]
ansible_python_interpreter=/usr/bin/python2.6
[root@python ~]# cat nginx.conf.j2
```

```
----- 此处省略 N 行 -----
worker_processes  {{ ansible_processor_cores }};
----- 此处省略 N 行 -----
```

关于 hosts 文件这里定义了一个主机组为 Nginx，组内包含 3 台设备，分别是 192.168.1.116、192.168.1.117、192.168.1.118，然后指定了 Nginx 组的一个变量 `ansible_python_interpreter`，因为目标机器上可能有多个 Python 版本，所以这里特意指定了一个 Python 版本去运行。关于 `nginx.conf.j2` 文件就是一个默认的 `nginx.conf` 文件，它只是针对 Nginx 的 `worker_processes` 参数通过 facts 信息中的 CPU 核心数目生成，其他的配置都是默认的。运行 playbook 之前我们需要确认 playbook 的语法信息是否正确。

对 `nginx.yaml` 使用 `--syntax-check` 参数 playbook 语法检测如下所示：

```
[root@python ~]# ansible-playbook nginx.yaml --syntax-check
playbook: nginx.yaml
```

使用 `--list-task` 参数显示 `nginx.yaml`，playbook 文件中所有的 task 名称如下所示：

```
[root@python ~]# ansible-playbook nginx.yaml --list-task
playbook: nginx.yaml
  play #1 (all):    TAGS: []
    Install Nginx Package  TAGS: []
    Copy Nginx.conf TAGS: []
```

使用 `--list-hosts` 参数显示 `nginx.yaml`，playbook 文件中针对的目标主机如下所示：

```
[root@python ~]# ansible-playbook nginx.yaml --list-hosts
playbook: nginx.yaml

  play #1 (all): host count=3
    192.168.1.116
    192.168.1.117
    192.168.1.118
```

确认信息都正确后，直接使用以下命令运行下 nginx.yaml playbook:

```
[root@python ~]# ansible-playbook -i hosts nginx.yaml -f 3
```

```
PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.118]
ok: [192.168.1.117]
ok: [192.168.1.116]

TASK: [Install Nginx Package] *****
changed: [192.168.1.116]
changed: [192.168.1.117]
changed: [192.168.1.118]

TASK: [Copy Nginx.conf] *****
changed: [192.168.1.118]
changed: [192.168.1.117]
changed: [192.168.1.116]

NOTIFIED: [ReStart Nginx Service] *****
changed: [192.168.1.116]
changed: [192.168.1.117]
changed: [192.168.1.118]

PLAY RECAP *****
192.168.1.116      : ok=4    changed=3    unreachable=0    failed=0
192.168.1.117      : ok=4    changed=3    unreachable=0    failed=0
192.168.1.118      : ok=4    changed=3    unreachable=0    failed=0
```

这样我们就完成了3台机器的Nginx安装部署。下面我们需要对主机的Nginx服务进行核查，并且确认生成后nginx.conf中的worker\_processes参数的值是否正确:

```
[root@python ~]# ansible -i hosts all -m shell -a 'netstat -tpln |grep
:80' -f 3
192.168.1.117 | success | rc=0 >>
tcp        0      0 0.0.0.0:80      0.0.0.0:*      LISTEN      2370/nginx
192.168.1.116 | success | rc=0 >>
tcp        0      0 0.0.0.0:80      0.0.0.0:*      LISTEN      6231/nginx
```

```

192.168.1.118 | success | rc=0 >>
tcp        0      0 0.0.0.0:80      0.0.0.0:*      LISTEN      17863/nginx
[root@python ~]# ansible -i hosts all -m shell -a 'grep worker_
processes /etc/nginx/nginx.conf' -f 3
192.168.1.117 | success | rc=0 >>
worker_processes 2;

192.168.1.116 | success | rc=0 >>
worker_processes 2;

192.168.1.118 | success | rc=0 >>
worker_processes 1;

```

后续如果又需要更改 Nginx 配置，只需修改 nginx.conf.j2 模板即可，在运行 playbook 的时候我们可以指定 task 运行，这个时候只运行 Copy Nginx.conf 即可：

```

[root@python ~]# ansible-playbook -i hosts nginx.yaml -f 3 --start-at-
task='Copy Nginx.conf'

```

```

PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.118]
ok: [192.168.1.117]
ok: [192.168.1.116]

TASK: [Copy Nginx.conf] *****
changed: [192.168.1.117]
changed: [192.168.1.118]
changed: [192.168.1.116]

NOTIFIED: [ReStart Nginx Service] *****
ok: [192.168.1.116]
ok: [192.168.1.117]
ok: [192.168.1.118]

PLAY RECAP *****
192.168.1.116      : ok=3    changed=1    unreachable=0    failed=0
192.168.1.117      : ok=3    changed=1    unreachable=0    failed=0
192.168.1.118      : ok=3    changed=1    unreachable=0    failed=0

```

当然, playbook 还支持交互式地执行 task, 我们可以指定 `--step` 参数即可。nginx.yml 是一个相对简单的 playbook 文件, 在我们的实际工作中可能会遇到各种复杂的需求, 但 playbook 的灵活性非常强大, 下面我们通过一个比较全面的 playbook 例子来讲解 playbook 的一些日常使用, 代码如下:

```

---
- hosts: 192.168.1.117:192.168.1.118      # 目标主机支持 'Ad-Hoc' 模式的所有
  patterns
  remote_user: root                       # 远程 SSH 认证用户
  sudo: yes                               # 设置 'playbook sudo' 操作
  sudo_user: yadmin                       # 设置 'playbook sudo' 用户
  gather_facts: no                        # 设置 'facts' 信息收集
  accelerate: no                          # 设置 'accelerate' 模式
  accelerate_port: 5099                  # 设置 'accelerate' 端口
  max_fail_percentage: 30                # 设置 'playbook tasks' 失败百分比
  connection: local                      # 设置远程连接方式
  serial: 15                             # 设置 'playbook' 并发数目
  vars:                                  # 设置 'playbook' 变量
    nginx_port: 80
  vars_files:                             # 设置 'playbook' 变量引用文件
    - "vars.yml"
    - [ "one.yml", "two.yml" ]
  vars_prompt:                            # 设置通过交互模式输入变量
    - name: "password vaes"
      prompt: "Enter password"           # 使用 'prompt' 模块加密输入变量
      default: "secret"
      private: yes
      encrypt: "md5_crypt"
      confirm: yes
      salt: 1234
      salt_size: 8
  pre_tasks:                             # 设置 'playbook' 运行之前的 'tasks'
    - name: pre_tasks
      shell: hostname
  roles:                                  # 设置引入 'role'
    - docker
    - { role: docker, version: '1.5.0', when: "ansible_system ==
      'Linux'", tags: [docker, install] }
    - { role: docker, when: ansible_all_ipv4_addresses ==
      '192.168.1.118' }

```

```

tasks:                                # 设置引入 'task'
- include: tasks.yaml
- include: tasks.yaml ansible_distribution='CentOS' ansible_
  distribution_version='6.6'
- { include: tasks.yaml, version: '1.1', package: [nginx,httpd]}
- include: tasks_192.168.1.117.yaml
  when: ansible_all_ipv4_addresses == '192.168.1.117'
post_tasks:                           # 设置 'playbook' 运行之后的 'tasks'
- name: post_tasks
  shell: hostname
handlers:                             # 设置 'playbooks' 的 'handlers'
- include: handlers.yaml

```

Ansible 的 playbook 写法很丰富，功能很强大，只有掌握了 playbook 每一个参数之后，我们才能写出强大而且灵活性很高的 playbook，这也是我们在工作中接触和使用最多的地方。

## 4.2 playbook 变量与引用

这节我们来讲解 playbook 变量，Ansible 定义变量的方式有很多种，下面就详细介绍 Ansible 各种变量定义方式。我们还可以针对主机或者主机组设置变量。

### 1. 通过 Inventory 文件定义主机以及主机组变量

首先我们来看下对应的 Inventory 文件，Ansible 默认的 Inventory 文件是 INI 格式，比如上一节用到的那个 hosts 文件，然后分别针对每台主机设置一个变量名称叫作 key，接着使用 debug 模块来查看变量的值，最后通过对 Nginx 组定义一个变量同样使用 debug 模块查看，如下所示：

```

192.168.1.116    key=116
192.168.1.117    key=117
192.168.1.118    key=118
[nginx]
192.168.1.11[6:8]
[nginx:vars]
ansible_python_interpreter=/usr/bin/python2.6

```

我们编写一个 playbook 文件来验证变量的引用是否正确：



```

---
- hosts: all
  gather_facts: False
  tasks:
    - name: diplay Host Variable from hostfile
      debug: msg="The {{ inventory_hostname }} Vaule is {{
        key }}"

```

我们运行这个 playbook 如下所示:

```
[root@python ~]# ansible-playbook variable.yaml
```

```

PLAY [all] *****

TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.116] => {
  "msg": "The 192.168.1.116 Vaule is 116"
}
ok: [192.168.1.117] => {
  "msg": "The 192.168.1.117 Vaule is 117"
}
ok: [192.168.1.118] => {
  "msg": "The 192.168.1.118 Vaule is 118"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

发现每台主机都引用了自己定义的变量, 接下来我们注释每台主机的变量定义, 直接给 nginx 组定义一个变量, 变量名称还是 key 且值为 nginx, 如下所示:

```

#192.168.1.116    key=116
#192.168.1.117    key=117
#192.168.1.118    key=118
[nginx]
192.168.1.11[6:8]
[nginx:vars]
ansible_python_interpreter=/usr/bin/python2.6
key=nginx

```

我们再来运行一下 playbook 文件:

```
[root@python ~]# ansible-playbook variable.yaml

PLAY [all] *****
TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.116] => {
    "msg": "The 192.168.1.116 Vaule is nginx"
}
ok: [192.168.1.118] => {
    "msg": "The 192.168.1.118 Vaule is nginx"
}
ok: [192.168.1.117] => {
    "msg": "The 192.168.1.117 Vaule is nginx"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0
```

因为所有主机都处于 nginx 组内, 所以该组定义的变量针对组内所有主机都生效。如果 nginx 组定义了变量, 然后每台主机也定义了变量, 只要定义的变量 key 名称不同, 我们都可以直接引用这些变量, 但是如果主机和主机组都定义了变量而且 key 还都相同, 这个时候你会发现单台主机定义的变量会生效, 大家可以自己进行测试。

## 2. 通过 /etc/ansible/ 下的文件定义主机以及主机组变量

默认使用 yum 安装 Ansible 的配置文件是在 /etc/ansible/ 目录下, 我们还可以使用在该目录下新建 host\_vars 和 group\_vars 目录来针对主机和主机组定义变量, 如果是采取其他方式安装的 Ansible 只需在 playbook 文件当前目录下新建这两个目录即可。如下所示目录和文件内容:

```
[root@python ansible]# tree
.
├── ansible.cfg
└── group_vars
```

```

|   └── nginx
|   └── hosts
|   └── host_vars
|       ├── 192.168.1.116
|       ├── 192.168.1.117
|       └── 192.168.1.118
2 directories, 6 files

[root@python ansible]# head host_vars/*
==> host_vars/192.168.1.116 <==
---
key: 192.168.1.116

==> host_vars/192.168.1.117 <==
---
key: 192.168.1.117

==> host_vars/192.168.1.118 <==
---
key: 192.168.1.118

[root@python ansible]# cat group_vars/nginx
---
key: NGINX

```

同样，我们运行上面的 playbook 文件来验证结果如下所示：

```

[root@python ansible]# ansible-playbook /root/variable.yaml

PLAY [all] *****

TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.117] => {
    "msg": "The 192.168.1.117 Vaule is 192.168.1.117"
}
ok: [192.168.1.116] => {
    "msg": "The 192.168.1.116 Vaule is 192.168.1.116"
}
ok: [192.168.1.118] => {

```

```

    "msg": "The 192.168.1.118 Vaule is 192.168.1.118"
  }
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

我们可以看到每台主机的变量已经生效，下面我们删掉 `host_vars` 下每台主机变量定义文件，然后来验证 `group_vars/` 下 `nginx` 组的变量定义：

```

[root@python ansible]# rm -fr host_vars/*
[root@python ansible]# ansible-playbook /root/variable.yaml

PLAY [all] *****

TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.116] => {
    "msg": "The 192.168.1.116 Vaule is NGINX"
}
ok: [192.168.1.117] => {
    "msg": "The 192.168.1.117 Vaule is NGINX"
}
ok: [192.168.1.118] => {
    "msg": "The 192.168.1.118 Vaule is NGINX"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

### 3. 通过 `ansible-playbook` 命令行传入

前面我们已经介绍了两种主机以及主机组变量的定义方式，这两种方式也是我们日常使用过程中最常用的两种变量定义方式。这节我们介绍一个通过 `ansible-playbook` 命令行传参的方式定义变量，但是默认传进去的变量都是全局变量，如下所示：

```

[root@python ~]# ansible-playbook /root/variable.yaml -e "key=KEY"

```

```
PLAY [all] *****
```

```
TASK: [display Host Variable from hostfile] *****
```

```
ok: [192.168.1.116] => {
```

```
  "msg": "The 192.168.1.116 Vaule is KEY"
```

```
}
ok: [192.168.1.117] => {
```

```
  "msg": "The 192.168.1.117 Vaule is KEY"
```

```
}
ok: [192.168.1.118] => {
```

```
  "msg": "The 192.168.1.118 Vaule is KEY"
```

```
}
PLAY RECAP *****
```

```
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
```

```
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
```

```
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0
```

当然也支持同时传多个变量，目前 Ansible-playbook 还支持指定文件的方式传入变量，变量文件的内容支持 YAML 和 JSON 两种格式，如下所示：

```
[root@python ~]# cat var.yaml
```

```
---
```

```
key: YAML
```

```
[root@python ~]# cat var.json
```

```
{"key": "JSON"}
```

这里我们指定 var.json 文件传入变量：

```
[root@python ~]# ansible-playbook /root/variable.yaml -e "@var.json"
```

```
PLAY [all] *****
```

```
TASK: [display Host Variable from hostfile] *****
```

```
ok: [192.168.1.116] => {
```

```
  "msg": "The 192.168.1.116 Vaule is JSON"
```

```
}
ok: [192.168.1.117] => {
```

```
  "msg": "The 192.168.1.117 Vaule is JSON"
```

```
}
ok: [192.168.1.118] => {
```



```

***** "msg": "The 192.168.1.118 Vaule is JSON"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

下面我们指定 var.yaml 文件传入变量：

```
[root@python ~]# ansible-playbook /root/variable.yaml -e "@var.yaml"
```

```

PLAY [all] *****

TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.116] => {
    "msg": "The 192.168.1.116 Vaule is YAML"
}
ok: [192.168.1.117] => {
    "msg": "The 192.168.1.117 Vaule is YAML"
}
ok: [192.168.1.118] => {
    "msg": "The 192.168.1.118 Vaule is YAML"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

#### 4. 在 playbook 文件内使用 vars

我们还可以在 playbook 文件内通过 vars 字段定义变量，再来看 variable.yaml 文件内容：

```

---
- hosts: all
  gather_facts: False
  vars:
    key: Ansible
  tasks:
    - name: diplay Host Variable from hostfile

```

```
debug: msg="The {{ inventory_hostname }} Vaule is {{
    key }}"
```

然后我们直接运行 variable.yaml 文件如下所示:

```
[root@python ~]# ansible-playbook /root/variable.yaml
```

```
PLAY [all] *****
```

```
TASK: [diplay Host Variable from hostfile] *****
```

```
ok: [192.168.1.117] => {
```

```
  "msg": "The 192.168.1.117 Vaule is Ansible"
```

```
}
```

```
ok: [192.168.1.116] => {
```

```
  "msg": "The 192.168.1.116 Vaule is Ansible"
```

```
}
```

```
ok: [192.168.1.118] => {
```

```
  "msg": "The 192.168.1.118 Vaule is Ansible"
```

```
}
```

```
PLAY RECAP *****
```

```
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
```

```
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
```

```
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0
```

## 5. 在 playbook 文件内使用 vars\_files

我们还可以在 playbook 文件内通过 vars\_files 字段引用变量, 首先把所有的变量定义到某个文件内, 然后在 playbook 文件内使用 vars\_files 参数引用这个变量文件, 我们再看 variable.yaml 文件的内容:

```
---
```

```
- hosts: all
```

```
  gather_facts: False
```

```
  vars_files:
```

```
    - var.yaml
```

```
  tasks:
```

```
    - name: diplay Host Variable from hostfile
```

```
      debug: msg="The {{ inventory_hostname }} Vaule is {{
        key }}"
```

var.yaml 文件就是变量定义存放的文件，这个时候我们就可以直接运行 variable.yaml，结果如下所示：

```
[root@python ~]# ansible-playbook /root/variable.yaml

PLAY [all] *****

TASK: [diplay Host Variable from hostfile] *****
ok: [192.168.1.116] => {
    "msg": "The 192.168.1.116 Vaule is YAML"
}
ok: [192.168.1.118] => {
    "msg": "The 192.168.1.118 Vaule is YAML"
}
ok: [192.168.1.117] => {
    "msg": "The 192.168.1.117 Vaule is YAML"
}

PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.117      : ok=1    changed=0    unreachable=0    failed=0
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0
```

## 6. 使用 register 内的变量

Ansible playbook 内 task 之间还可以互相传递数据，比如我们总共有两个 tasks，其中第 2 个 task 是否执行是需要判断第 1 个 task 运行后的结果，这个时候我们就得在 task 之间传递数据，需要把第 1 个 task 执行的结果传递给第 2 个 task。Ansible task 之间传递数据使用 register 方式，下面我们来看一个简单的例子：

```
---
- hosts: all
  gather_facts: False
  tasks:
    - name: register variable
      shell: hostname
      register: info
    - name: display variable
      debug: msg="The varibale is {{ info }}"
```

这里我们把第 1 个 task 执行 hostname 的结果 register 给 info 这个变量，然后在第

2 个 task 把这个结果使用 debug 模块打印出来, 先来看执行结果, 如下所示:

```
[root@python ~]# ansible-playbook variable.yaml -l 192.168.1.118

PLAY [all] *****

TASK: [register variable] *****
changed: [192.168.1.118]

TASK: [display variable] *****
ok: [192.168.1.118] => {
  "msg": "The varibale is {u'changed': True, u'end': u'2015-05-31 16:09:42.511109', u'stdout': u'python', u'cmd': u'hostname', u'rc': 0, u'start': u'2015-05-31 16:09:42.506714', u'stderr': u'', u'delta': u'0:00:00.004395', 'invocation': {'module_name': u'shell', 'module_args': u'hostname'}, 'stdout_lines': [u'python'], u'warnings': []}"
}

PLAY RECAP *****
192.168.1.118      : ok=2    changed=1    unreachable=0    failed=0
```

我们可以看到 info 的结果是一段 Python 字典数据, 里面存储着很多信息包括执行时间状态变化输出等信息。register 的输出数据结果都是 Python 字典, 我们可以很容易地挑选出我们想要的信息, 比如下面想标准输出 stdout 的信息时, 只需要指定 stdout 这个 key 即可, 如下所示:

```
[root@python ~]# cat variable.yaml
---
- hosts: all
  gather_facts: False
  tasks:
    - name: register variable
      shell: hostname
      register: info
    - name: display variable
      debug: msg="The varibale is {{ info['stdout'] }}"
```

info['stdout'] 是一个标准的 Python 语言在字典中取值的用法, 我们再来执行 playbook 如下所示:

```
[root@python ~]# ansible-playbook variable.yaml -l 192.168.1.118
```

```
PLAY [all] *****

TASK: [register variable] *****
changed: [192.168.1.118]

TASK: [display variable] *****
ok: [192.168.1.118] => {
  "msg": "The varibale is python"
}

PLAY RECAP *****
192.168.1.118 : ok=2    changed=1    unreachable=0    failed=0
```

这个时候我们就只取出 stdout 的值了，其他信息的引用与这个方式一样。

## 7. 使用 vars\_prompt 传入

Ansible 还支持在运行 playbook 的时候通过交互式的方式给定义好的参数传入变量值，只需在 playbook 中定义 vars\_prompt 的变量名和交互式提示内容即可。当然 Ansible 还可以对输入的变量值进行加密处理，比如采用 SHA512 和 MD5 算法加密。需要注意的是，如果要对变量值进行加密的话，Ansible 机器上需要安装 passlib python 库。下面我们通过一个简单的例子来了解 vars\_prompt 交互式传入变量值：

```
---
- hosts: all
  gather_facts: False
  vars_prompt:
    - name: "one"
      prompt: "please input one value"
      private: no
    - name: "two"
      prompt: "please input two value"
      default: 'good'
      private: yes
  tasks:
    - name: display one value
      debug: msg="one value is {{ one }}"
    - name: display two value
      debug: msg="two value is {{ two }}"
```



在例子中通过 `vars_prompt` 参数进行交互传入两个变量的值，变量名分别是 `one` 和 `two`，`one` 变量定义为非私有变量，`two` 变量定义为私有变量并且还提供一个默认值。如果不给变量 `two` 传入值的话，`two` 变量的值将会为默认值。`private: yes` 和 `private: no` 的作用是显示交互模式下输入的变量值。我们来运行 `playbook` 测试一下：

```
[root@python ~]# ansible-playbook variable.yaml -l 192.168.1.118

please input one value: ansible
please input two value [good]:

PLAY [all] *****

TASK: [display one value] *****
ok: [192.168.1.118] => {
    "msg": "one value is ansible"
}

TASK: [display two value] *****
ok: [192.168.1.118] => {
    "msg": "two value is ok"
}

PLAY RECAP *****
192.168.1.118      : ok=2    changed=0    unreachable=0    failed=0
```

前面我们介绍了 7 种方式去定义变量以及如何引用。变量的定义有很多方式，上面介绍的是常用的几种方式。Ansible 的变量引用方式相对少，都是固定的 `{{ key }}` 或者 `{{ key['key'] }}` 或者 `{{ key[0]['key'] }}` 根据变量的值定义了不同的数据结构，因而变量引用方法稍微有点区别。与 Python 语言引用方式一样，还有一个要注意的是，如果我们通过多种方式定义了相同变量，变量的名称都是一样的，但是每种方式定义的值不一样，这个时候大家会问到底哪个会生效或者哪个会被覆盖。建议读者在掌握各种变量定义方式后进行测试，本书就不进行相应测试了。

### 4.3 playbook 循环

有时候我们写 `playbook` 的时候发现写了很多的 `task` 都重复引用某个模块，比如一次想同步 10 个文件，如果按照以前写 `playbook` 的思路需要写 10 个 `task`，这样写的话发现

playbook 会显得很臃肿。这节我们就介绍 Ansible loops 用法，可以使用 loops 方式去编写 playbook 减少重复使用某个模块。目前官网支持很多 loops 方式，由于篇幅有限所以本书只挑选了几个比较常用的 loops 进行介绍，关于其他的 loops 大家可以通关官网 [http://docs.ansible.com/ansible/playbooks\\_loops.html#standard-loops](http://docs.ansible.com/ansible/playbooks_loops.html#standard-loops) 进行更加详细的了解。

## 1. 标准 Loops

标准 loops 是我们在编写 playbook 过程中使用最多的一种 loops，它能直接减少编写 task 的次数，比如需要使用 yum 模块安装 10 个软件包，按照以前的思路可能需要写 10 个 task 每个 task 使用 yum 安装每个软件包，这个时候我们就可以直接使用标准的 loops 简单快速地实现 10 个软件包的安装了。比如下面的例子分别打印 one two 这两个值，如下所示：

```
---
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      debug: msg="name -----> {{ item }}"
      with_items:
        - one
        - two
```

运行 loops.yaml，如下所示：

```
[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116

PLAY [all] *****

TASK: [debug loops] *****
ok: [192.168.1.116] => (item=one) => {
  "item": "one",
  "msg": "name -----> one"
}
ok: [192.168.1.116] => (item=two) => {
  "item": "two",
  "msg": "name -----> two"
}

PLAY RECAP *****
```

```
192.168.1.116 : ok=1    changed=0    unreachable=0    failed=0
```

with\_items 的值是 python list 数据结构，可以理解为每个 task 会循环读取 list 里面的值，然后 key 的名称是 item，当然 list 里面也支持 python 字典，如下所示：

```
---
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      debug: msg="name -----> {{ item.key }}    vaule ----
        ----> {{ item.vaule }}"
      with_items:
        - {key: "one", vaule: "VAULE1"}
        - {key: "two", vaule: "VAULE2"}
```

运行 loops.yaml 如下：

```
[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116
```

```
PLAY [all] *****
```

```
TASK: [debug loops] *****
```

```
ok: [192.168.1.116] => (item={'vaule': 'VAULE1', 'key': 'one'}) => {
```

```
  "item": {
    "key": "one",
    "vaule": "VAULE1"
```

```
  },
```

```
  "msg": "name -----> one    vaule -----> VAULE1"
```

```
}
```

```
ok: [192.168.1.116] => (item={'vaule': 'VAULE2', 'key': 'two'}) => {
```

```
  "item": {
    "key": "two",
    "vaule": "VAULE2"
```

```
  },
```

```
  "msg": "name -----> two    vaule -----> VAULE2"
```

```
}
```

```
PLAY RECAP *****
```

```
192.168.1.116 : ok=1    changed=0    unreachable=0    failed=0
```

## 2. 嵌套 Loops

嵌套 Loops 也是我们编写 playbook 中比较常见的一种循环，它主要实现一对多或者多对多的合并，如下所示：

```
---
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      debug: msg="name -----> {{ item[0] }}      vaule -----> {{ item[1] }}"
      with_nested:
        - ['A']
        - ['a','b','c']
```

运行 loops.yaml 如下所示：

```
[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116

PLAY [all] *****
TASK: [debug loops] *****
ok: [192.168.1.116] => (item=['A', 'a']) => {
  "item": [
    "A",
    "a"
  ],
  "msg": "name -----> A      vaule -----> a"
}
ok: [192.168.1.116] => (item=['A', 'b']) => {
  "item": [
    "A",
    "b"
  ],
  "msg": "name -----> A      vaule -----> b"
}
ok: [192.168.1.116] => (item=['A', 'c']) => {
  "item": [
    "A",
    "c"
  ],
  "msg": "name -----> A      vaule -----> c"
}
```

```

"msg": "name -----> A   vaule -----> c"
}

PLAY RECAP *****
192.168.1.116: ok=1 changed=0 unreachable=0 failed=0

```

我们可以通过一个 Python 例子来解释这个例子，定义两个 list 的代码如下：

```

In [12]: one=['A']
In [13]: two=['a','b','c']
In [14]: [i + y for i in one for y in two]
Out[14]: ['Aa', 'Ab', 'Ac']

```

### 3. 散列 loops

散列 loops 相比标准 loops 就是变量支持更丰富的数据结构，比如标准 loops 的最外层数据必须是 Python 的 list 数据类型，而散列 loops 直接支持 YAML 格式的数据变量。下面我们通过一个简单的例子来了解散列 loops，如下所示：

```

---
- hosts: all
  gather_facts: False
  vars:
    user:
      shencan:
        name: shencan
        shell: bash
      ruifengyun:
        name: ruifengyun
        shell: zsh
  tasks:
    - name: debug loops
      debug: msg="name -----> {{ item.key }}   vaule ---
        -----> {{ item.value.name }}   shell -----> {{
          item.value.shell }}"
      with_dict: user

```

运行 loops.yaml 如下所示：

```

[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116

```



```

PLAY [all] *****

TASK: [debug loops] *****
ok: [192.168.1.116] => (item={'key': 'ruifengyun', 'value': {'shell':
'zsh', 'name': 'ruifengyun'}}) => {
  "item": {
    "key": "ruifengyun",
    "value": {
      "name": "ruifengyun",
      "shell": "zsh"
    }
  },
  "msg": "name -----> ruifengyun    vaule -----> ruifengyun
        shell -----> zsh"
}
ok: [192.168.1.116] => (item={'key': 'shencan', 'value': {'shell':
'bash', 'name': 'shencan'}}) => {
  "item": {
    "key": "shencan",
    "value": {
      "name": "shencan",
      "shell": "bash"
    }
  },
  "msg": "name -----> shencan    vaule -----> shencan shell ---
        -----> bash"
}

PLAY RECAP *****
192.168.1.116: ok=1 changed=0 unreachable=0 failed=0

```

with\_dict 是接收一个 Python 字典（经过 yaml.load 后）的格式的变量，为了更好地理解，下面我们通过 Python 语言实现的这个过程：

```

In [14]: user
Out[14]:
{'ruifengyun': {'name': 'ruifengyun', 'shell': 'zsh'},
 'shencan': {'name': 'shencan', 'shell': 'bash'}}

```

```

In [15]: for key,value in user.items():

```

```

.....:   print key,value['name'],value['shell']
.....:
ruifengyun ruifengyun zsh
shencan shencan bash

```

#### 4. 文件匹配 loops

文件匹配 loops 是我们编写 playbook 的时候需要针对文件进行操作中最常用的一种循环,比如我们需要针对一个目录下指定格式的文件进行处理,这个时候直接在引用 with\_fileglob 循环去匹配我们需要处理的文件即可,下面我们通过例子进行讲解,如下所示:

```

---
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      debug: msg="files -----> {{ item }}"
      with_fileglob:
        - /root/*.yaml

```

with\_fileglob 这个时候会匹配 root 目录下所有以 yaml 结尾的文件,当作 {{ item }} 变量,运行结果如下:

```

[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116

PLAY [all] *****

TASK: [debug loops] *****
ok: [192.168.1.116] => (item=/root/nginx.yaml) => {
  "item": "/root/nginx.yaml",
  "msg": "files -----> /root/nginx.yaml"
}
ok: [192.168.1.116] => (item=/root/loops.yaml) => {
  "item": "/root/loops.yaml",
  "msg": "files -----> /root/loops.yaml"
}
ok: [192.168.1.116] => (item=/root/variable.yaml) => {
  "item": "/root/variable.yaml",
  "msg": "files -----> /root/variable.yaml"
}

```

```
ok: [192.168.1.116] => (item=/root/var.yaml) => {
  "item": "/root/var.yaml",
  "msg": "files -----> /root/var.yaml"
}
```

```
PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
```

其实文件 loops 就是使用 python glob 模块去做文件模糊匹配，如下所示：

```
In [1]: import glob
```

```
In [2]: print glob.glob('/root/*.yaml')
```

```
['/root/nginx.yaml', '/root/loops.yaml', '/root/variable.yaml', '/root/
var.yaml']
```

## 5. 随机选择 loops

随机选择 loops 的例子如下：

```
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      debug: msg="name -----> {{ item }}"
      with_random_choice:
        - "ansible1"
        - "ansible2"
        - "ansible3"
```

运行 loops.yaml 如下所示：

```
[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.116
```

```
PLAY [all] *****
```

```
TASK: [debug loops] *****
```

```
ok: [192.168.1.116] => (item=ansible2) => {
  "item": "ansible2",
  "msg": "name -----> ansible2"
}
```

```
PLAY RECAP *****
192.168.1.116      : ok=1    changed=0    unreachable=0    failed=0
```

with\_random\_choice 就是在传入的 list 中随机选择一个，与使用 python random 实现原理一样，如下所示：

```
In [5]: import random
```

```
In [6]: list=['ansible1','ansible2','ansible2']
```

```
In [7]: print(random.choice(list))
ansible1
```

## 6. 条件判断 Loops

有时候执行一个 task 之后，我们需要检测这个 task 的结果是否达到了预想状态，如果没有达到我们预想的状态时，就需要退出整个 playbook 执行，这个时候我们就需要对某个 task 结果一直循环检测了，如下所示：

```
---
- hosts: all
  gather_facts: False
  tasks:
    - name: debug loops
      shell: cat /root/Ansible
      register: host
      until: host.stdout.startswith("Master")
      retries: 5
      delay: 5
```

5 秒执行一次 cat /root/Ansible 将结果 register 给 host 然后判断 host.stdout.startswith 的内容是否是 Master 字符串开头，如果条件成立，此 task 运行完成，如果条件不成立 5 秒后重试，5 次后还不成立，此 task 运行失败。

## 7. 文件优先匹配 Loops

在前面我们介绍了一个文件匹配 loops，其实文件优先匹配 loops 和它的功能很相似，因为都是用来做文件的匹配，但是文件优先匹配会根据你传入的变量或者文件进行从上往下匹配，如果匹配到某个文件，它会用这个文件当作 {{ item }} 的值，如下所示：

```
---
- hosts: all
  gather_facts: True
```

```

tasks:
  - name: debug loops
    debug: msg="files -----> {{ item }}"
    with_first_found:
      - "{{ ansible_distribution }}.yaml"
      - "default.yaml"

```

with\_first\_found 会从 list 里面定义的文件从上往下一个一个的匹配，如果匹配到了 item 就是该文件，如下所示：

```

[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.118

PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.118]

TASK: [debug loops] *****
ok: [192.168.1.118] => (item=/root/CentOS.yaml) => {
  "item": "/root/CentOS.yaml",
  "msg": "files -----> /root/CentOS.yaml"
}

PLAY RECAP *****
192.168.1.118      : ok=2    changed=0    unreachable=0    failed=0

```

## 8. register Loops

在上一小节介绍 playbook 变量定义的时候，我们已经知道 register 是用于 task 直接互相传递数据的，一般我们会把 register 用在单一的 task 中进行变量临时存储，其实 register 还可以同时接受多个 task 的结果当作变量临时存储，如下所示：

```

---
- hosts: all
  gather_facts: True
  tasks:
    - name: debug loops
      shell: "{{ item }}"
      with_items:
        - hostname
        - uname

```



```

register: ret
- name: display loops
  debug: msg="{% for i in ret.results %} {{ i.stdout }} {%
    endfor %}"

```

以前我们写 playbook 的时候都是执行一个 task，然后 register 给一个变量，它的数据结果很好理解，但是如果你执行多个 task 并且 register 给一个变量时，它的结果数据跟平常就不一样了，比如上面我们需要使用 jinja2 的 for 循环才能把所有的结果显示出来，如下所示：

```

[root@python ~]# ansible-playbook loops.yaml -l 192.168.1.118

PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.118]

TASK: [debug loops] *****
changed: [192.168.1.118] => (item=hostname)
changed: [192.168.1.118] => (item=uname)

TASK: [display loops] *****
ok: [192.168.1.118] => {
  "msg": " python  Linux "
}

PLAY RECAP *****
192.168.1.118      : ok=3    changed=1    unreachable=0    failed=0

```

前面我们已经介绍了常用的一些 Ansible loops 以及通过示例去了解它的实现原理。当然如果你有 Python 开发能力，就可以去看 Ansible 源码，这样可以对每一个 loops 的实现原理进行了解，还可以自己编写 loops。

## 4.4 playbook lookups

通过第 2 小节的学习我们知道了有很多方式可以定义 Ansible 变量，但是这些变量的定义都是静态的。其实 Ansible 还支持从外部数据拉取信息，比如我们可以从数据库里面读取信息然后定义给一个变量的形式，这就是 Ansible 的 lookups 插件。目前

Ansible 已经自带一些 lookups 组件，我们可以从 Ansible 源码文件中查看，本节会挑选几个经常使用的 lookups 进行讲解，还有目前所有的 lookups 都是在控制机上运行的，有一些软件依赖需要注意。

## 1. lookups file

file 是我们经常使用的一种 lookups 方式，它的原理就是使用 Python 的 codecs.open 打开文件然后把结果返回给变量，下面我们来编写一个 playbook 然后了解整个使用过程，如下所示：

```
---
- hosts: all
  gather_facts: False
  vars:
    contents: "{{ lookup('file', '/etc/sysconfig/network') }}"
  tasks:
    - name: debug lookups
      debug: msg="The contents is {% for i in contents.
        split("\n") %} {{ i }} {% endfor %}"
```

为了 codecs.open 读取文件后格式化更加友好的显示，这里使用了 jinja 模板进行了相应的格式化，下面我们运行这个 playbook：

```
[root@python ~]# ansible-playbook lookups.yaml -l 192.168.1.118

PLAY [all] *****

TASK: [debug lookups] *****
ok: [192.168.1.118] => {
  "msg": "The contents is NETWORKING=yes HOSTNAME=python "
}

PLAY RECAP *****
192.168.1.118: ok=1 changed=0 unreachable=0 failed=0
```

## 2. lookups password

password 也是我们经常使用的一种 lookups 方式，它会对传入的内容进行加密处理，如下所示：

```
---
- hosts: all
```

```

gather_facts: False
vars:
  contents: "{{ lookup('password', 'ansible_book') }}"
tasks:
  - name: debug lookups
    debug: msg="The contents is {{ contents }}"

```

运行 playbook 就可以看到加密后的字符了:

```
[root@python ~]# ansible-playbook lookups.yaml -l 192.168.1.118
```

```

PLAY [all] *****

TASK: [debug lookups] *****
ok: [192.168.1.118] => {
  "msg": "The contents is Np6m0b9ZqrzVU3Sux5gH"
}

PLAY RECAP *****
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

### 3. lookups pipe

pipe lookups 的实现原理很简单, 如果阅读过源码的读者能发现它其实就是在控制机器上调用 subprocess.Popen 执行命令, 然后将命令的结果传递给变量, 如下所示:

```

---
- hosts: all
  gather_facts: False
  vars:
    contents: "{{ lookup('pipe', 'date +%Y-%m-%d') }}"
  tasks:
    - name: debug lookups
      debug: msg="The contents is {% for i in contents.
        split("\n") %} {{ i }} {% endfor %}"

```

contents 的内容就是在 Ansible 控制机器上执行 date +%Y-%m-%d 的结果:

```
[root@python ~]# ansible-playbook lookups.yaml -l 192.168.1.118
```

```
PLAY [all] *****
```

```

TASK: [debug lookups] *****
ok: [192.168.1.118] => {
    "msg": "The contents is 2015-05-31 "
}

PLAY RECAP *****
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0

```

#### 4. lookups redis\_kv

redis\_kv 是从 Redis 数据库中 get 数据，因为需要中控机连接 Redis 所以需要安装 Redis Python 库，使用 pip 方式安装就行，这里在本地安装了一个 Redis 服务器，然后给 Ansible 设置了一个值，如下所示：

```

[root@python ~]# redis-cli
redis 127.0.0.1:6379> set 'ansible' "good"
OK
redis 127.0.0.1:6379> get ansible
"good"
redis 127.0.0.1:6379>

```

然后我们修改 playbook 再运行，如下所示：

```

---
- hosts: all
  gather_facts: False
  vars:
    contents: "{{ lookup('redis_kv', 'redis://localhost:6379,ansible') }}"
  tasks:
    - name: debug lookups
      debug: msg="The contents is {% for i in contents.split("\n") %} {{ i }} {% endfor %}"

```

```

[root@python ~]# ansible-playbook lookups.yaml -l 192.168.1.118

```

```

PLAY [all] *****

TASK: [debug lookups] *****
ok: [192.168.1.118] => {
    "msg": "The contents is good "
}

```

```
}

```

```
PLAY RECAP *****
192.168.1.118      : ok=1    changed=0    unreachable=0    failed=0
```

## 5. lookups template

template 跟 file 方式有点类似，都是读取文件，但是 template 在读取文件之前需要把 jinja 模板渲染完后再读取，下面我们指定一个 jinja 模板文件：

```
worker_processes {{ ansible_processor_cores }};
IPaddress {{ ansible_eth0.ipv4.address }}
```

然后修改 playbook 如下所示：

```
---
- hosts: all
  gather_facts: True
  vars:
    contents: "{{ lookup('template', './lookups.j2') }}"
  tasks:
    - name: debug lookups
      debug: msg="The contents is {% for i in contents.
split("\n") %} {{ i }} {% endfor %}"
```

需要注意的是，lookups.j2 里面定义的变量是每台主机自己的 facts 信息，不是控制机的 facts 信息，运行结果如下：

```
[root@python ~]# ansible-playbook lookups.yaml

PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.118]
ok: [192.168.1.117]
ok: [192.168.1.116]

TASK: [debug lookups] *****
ok: [192.168.1.116] => {
  "msg": "The contents is  worker_processes 2;  IPaddress
192.168.1.116  "
}
```



```

ok: [192.168.1.117] => {
  "msg": "The contents is worker_processes 2; IPAddress
192.168.1.117 "
}
ok: [192.168.1.118] => {
  "msg": "The contents is worker_processes 1; IPAddress
192.168.1.118 "
}

PLAY RECAP *****
192.168.1.116: ok=2 changed=0 unreachable=0 failed=0
192.168.1.117: ok=2 changed=0 unreachable=0 failed=0
192.168.1.118: ok=2 changed=0 unreachable=0 failed=0

```

本节只是介绍了几种比较常用的 lookups 方式, Ansible 其他 lookups 方式还有很多, 读者可去阅读 Ansible 源码或者从官网文档进行相应的了解, 如果你有 Python 开发能力的话还可以编写适合自己需求的 lookups 方式。

## 4.5 playbook conditionals

本节介绍 conditionals, 在第 2 小节我们介绍 Ansible tasks 之间通过 register 进行数据的传递的时候简单提到过 conditionals, 而在实际应用过程中经常会碰到不同的主机可能要执行不同的命令, 或者执行某个 task 的时候需要进行一个简单的逻辑判断, 此刻就需要在写 task 的时候进行相应的判断。目前 Ansible 的所有 conditionals 方式都是使用 when 进行判断, when 的值是一个条件表达式, 如果条件判断成立, 这个 task 就执行某个操作, 如果条件判断不成立, 该 task 不执行或者某个操作会跳过。这里所说的成立与不成立就是 Python 语言里面的 True 与 False, 关于这个条件表达式也支持多个条件之间 and 或者 or, 还需要注意的是, 如果我们使用一个变量进行相应的判断, 一定要清楚该变量的数据类型。下面我们通过一个例子介绍常用的 conditionals 表达式, 如下所示:

```

---
- hosts: all
  tasks:
    - name: Host 192.168.1.118 run this task
      debug: msg="{{ ansible_default_ipv4.address }}"

```

```

1 when: ansible_default_ipv4.address == "192.168.1.118"
  - name: memtotal < 500M and processor_cores == 2 run this task
    debug: msg="{{ ansible_fqdn }}"
2 when: ansible_memtotal_mb < 500 and ansible_processor_cores == 2
  - name: all host run this task
    shell: hostname
    register: info

  - name: Hostname is python Machie run this task
    debug: msg="{{ ansible_fqdn }}"
3 when: info['stdout'] == "python"
  - name: Hostname is startswith M run this task
    debug: msg="{{ ansible_fqdn }}"
4 when: info['stdout'].startswith('M')

```

- 第1个 when 是判断 facts 信息，因为 `ansible_default_ipv4` 的数据结构是一个 Python 字典，`ansible_default_ipv4.address` 是取 IP 地址然后与 "192.168.1.118" 进行判断，这个判断是 Python 语法的判断，还有 `ansible_default_ipv4.address` 的值是 Python 的 str 数据类型，所以一定要用引号 "192.168.1.118"。
- 第2个 when 是判断 facts 的 `ansible_memtotal_mb` 和 `ansible_processor_cores` 信息，因为这两个信息的值都是 Python 的 int 数据类型，所有这里就不需要引号了，然后两个条件表达式用 `and` 结合。
- 第3个 when 是判断第3个 task 的运行结果 `stdout` 的值，前面我们已经介绍过 `register` 的数据结构了。
- 第4个 when 也是判断第3个 task 的运行结果 `stdout` 的值，这里使用了 Python 的 str 内置方法 `startswith`，因为 `startswith` 的方法最终会返回 `True` 和 `False`，所以这这也是一个条件表达式。

下面我们来执行这个 playbook:

```
[root@python ~]# ansible-playbook conditionals.yaml
```

```

PLAY [all] *****
*****
GATHERING FACTS *****

```

```

ok: [192.168.1.118]
ok: [192.168.1.117]
ok: [192.168.1.116]

TASK: [Host 192.168.1.118 run this task] *****
skipping: [192.168.1.116]
skipping: [192.168.1.117]
ok: [192.168.1.118] => {
    "msg": "192.168.1.118"
}

TASK: [memtotal < 500M and processor_cores == 2 run this task]
skipping: [192.168.1.116]
skipping: [192.168.1.118]
ok: [192.168.1.117] => {
    "msg": "Minion"
}

TASK: [all host run this task] *****
changed: [192.168.1.117]
changed: [192.168.1.116]
changed: [192.168.1.118]

TASK: [Hostname is python Machie run this task] *****
skipping: [192.168.1.116]
skipping: [192.168.1.117]
ok: [192.168.1.118] => {
    "msg": "python"
}

TASK: [Hostname is startswith M run this task] *****
skipping: [192.168.1.118]
ok: [192.168.1.116] => {
    "msg": "Master"
}
ok: [192.168.1.117] => {
    "msg": "Minion"
}

PLAY RECAP *****

```

```

192.168.1.116      : ok=3    changed=1    unreachable=0    failed=0
192.168.1.117      : ok=4    changed=1    unreachable=0    failed=0
192.168.1.118      : ok=4    changed=1    unreachable=0    failed=0

```

skipping 表示该 task 本机没有执行，整个过程可以清楚地看到每个 task 在哪台机器执行了，哪台机器没有执行。

## 4.6 Jinja2 filter

如果使用过 Jinja2 模板，肯定对 Jinja2 的 filter 不陌生，Jinja2 是目前比较流行的一款模板语言，Ansible 和 SaltStack 这两个配置管理工具都是只把它当作默认的模板语言。Ansible 默认支持 Jinja2 语言的内置 filter，Jinja2 官网也提供了很多 filter，建议读者阅读，本节会选择几个经常使用的 filter 进行介绍，如下所示：

```

---
- hosts: all
  gather_facts: False
  vars:
    list: [1,2,3,4,5]
    one: "1"
    str: "string"
  tasks:
    - name: run commands
      shell: df -h
      register: info

    - name: debug pprint filter
1      debug: msg="{{ info.stdout | pprint }}"

    - name: debug conditionals filter
2      debug: msg="The run commands status is changed"
      when: info|changed

    - name: debug int capitalize filter
3      debug: msg="The int value {{ one | int }} The lower value
        is {{ str | capitalize }}"

    - name: debug default filter
4      debug: msg="The Variable value is {{ ansible | default('ansible

```

```

is not define') }}"
- name: debug list max and min filter
5   debug: msg="The list max value is {{ list | max }} The list
      min value is {{ list | min }}"
- name: debug random filter
6   debug: msg="The list random value is {{ list | random }} and
      generate a random value is {{ 1000 | random(1, 10) }}"
- name: debug join filter
7   debug: msg="The join filter value is {{ list | join("+") }}"
- name: debug replace and regex_replace filter
8   debug: msg="The replace value is {{ str | replace('t','T') }}
      The regex_replace value is {{ str | regex_replace('.*tr(.*)$',
      '\\1') }}"

```

- 第 1 个是对 info.stdout 结果使用 pprint filter 进行格式化。
- 第 2 个是对 info 的执行状态使用 changed filter 进行判断。
- 第 3 个是对 one 的值进行 int 转变，然后对 str 的值进行 capitalize 格式化。
- 第 4 个是对 Ansible 变量进行判断，如果该变量定义了就引用它的值，如果没有定义就使用 default 内值。
- 第 5 个是对 list 内的值进行最大值 max 和最小值 min 取值。
- 第 6 个是对 list 内的值使用 random filter 随机挑选一个，然后随机生成 1000 以内的数字，step 是 10。
- 第 7 个是对 list 内的值使用 join filter 连接在一起。
- 第 8 个是对 str 的值使用 replace 与 regex\_replace 替换。

下面来运行 playbook，看看结果：

```
[root@python ~]# ansible-playbook filter.yaml -l 192.168.1.118
```

```

PLAY [all] *****
TASK: [run commands] *****
changed: [192.168.1.118]

```



TASK: [debug to\_nice\_yaml filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": " u'Filesystem Size    Used Avail Use% Mounted on\\n/
dev/mapper/VolGroup-lv_root    6.7G    1.9G    4.4G    31%  /\\ntmpfs
246M 0    246M    0% /dev/shm\\n/dev/sda1 485M 33M    427M 8% /boot'"
}
```

TASK: [debug conditionals filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The run commands status is changed"
}
```

TASK: [debug int capitalize filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The int value 1 The lower value is String"
}
```

TASK: [debug default filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The Variable value is ansible is not define"
}
```

TASK: [debug list max and min filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The list max value is 5 The list min value is 1"
}
```

TASK: [debug random filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The list random value is 2 and generate a random value is 281"
}
```

TASK: [debug join filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
  "msg": "The join filter value is 1+2+3+4+5"
}
```

TASK: [debug replace and regex\_replace filter] \*\*\*\*\*

```
ok: [192.168.1.118] => {
```

```

    "msg": "The replace value is sString The regex_replace vaule is ing "
  }

```

```

PLAY RECAP *****
192.168.1.118      : ok=9    changed=1    unreachable=0    failed=0

```

## 4.7 playbook 内置变量

playbook 默认已经内置变量，掌握了这些变量之后，我们就可以很容易实现关于主机相关的逻辑判断了。本节挑选了 7 个经常使用的内置变量进行详细讲解，最后我们也会通过一个案例告诉大家如何去引用这些变量。

### 1. groups 和 group\_names

groups 变量是一个全局变量，它会打印出 Inventory 文件里面的所有主机以及主机组信息，它返回的是一个 JSON 字符串，我们可以直接把它当作一个变量使用 {{ groups }} 格式进行调用。当然我们还可以引用 {{ groups }} 字符串里面的数据，比如引用 docker 组的 host，使用 {{ groups['docker'] }} 方式引用就行，它会返回一个主机 list 列表，group\_names 变量会打印当前主机所在的 groups 名称。如果没有定义会返回 ungrouped，它返回的也是一个组名称的 list 列表。

### 2. hostvars

hostvars 是用来调用指定主机变量，需要传入主机信息，返回结果也是一个 JSON 字符串，同样，也可以直接引用 JSON 字符串内的指定信息。

### 3. inventory\_hostname 和 inventory\_hostname\_short

inventory\_hostname 变量是返回 Inventory 文件里面定义的主机名，inventory\_hostname\_short 会返回 Inventory 文件中主机名的第一部分。

### 4. play\_hosts 和 inventory\_dir

play\_hosts 变量是用来返回当前 playbook 运行的主机信息，返回格式是主机 list 结构，inventory\_dir 变量是返回当前 playbook 使用的 Inventory 目录。

### 5. 应用案例

我们先来看一个 Jinja2 的模板文件，在这个模板文件里面使用上面介绍的所有

playbook 内置变量，如下所示：

```
=====
groups info
{{ groups }}
=====

=====docker groups info =====
{% for host in groups['docker'] %}

-={{ hostvars[host]['inventory_hostname'] }} eth0 IP is {{
  hostvars[host]['ansible_default_ipv4']['address'] }}

+={{ hostvars[host]['inventory_hostname'] }} groups is {{ group_names
  }}

_ {{ hostvars[host]['inventory_hostname'] }} short is {{ inventory_
  hostname_short }}

*****PLAY [all]*****
*{{ play_hosts }}
*****GATHERING FACTS*****
@{{ inventory_dir }}
{% endfor %}
```

这里先不用关注这个模板文件的作用，在我们运行完成后根据生成后文件结果就会对这个 Jinja 模板的结构很熟悉了。首先来看一下我们的 Inventory 信息，这里采用了多个 Inventory 文件，如下所示：

```
root@Master inventory)# tree .
```

```
.
├── docker
└── hosts
```

```
0 directories, 2 files
```

```
[root@Master inventory]# cat docker
```

```
[docker]
```

```
172.17.42.10[1:3]
```

```
[docker:vars]
```

```
ansible_ssh_pass='123456'
```

```
[ansible:children]
```

```
docker
```

```
[root@Master inventory]# cat hosts
172.17.42.101 ansible_ssh_pass='123456'
172.17.42.102 ansible_ssh_pass='123456'
172.17.42.1   ansible_ssh_pass='123456'
```

我们简单定义一个渲染 Jinja 模板的 playbook:

```
[root@Master ~]# cat template.yaml
---
- hosts: all
  tasks:
    - name: test template
      template: src=jinja.j2 dest=/tmp/cpis
```

jinja.j2 就是上面定义的 Jinja 模板文件，最后我们来执行这个 playbook:

```
[root@Master ~]# ansible-playbook template.yaml
```

```
PLAY [all] *****

GATHERING FACTS *****
ok: [172.17.42.102]
ok: [172.17.42.103]
ok: [172.17.42.101]
ok: [172.17.42.1]

TASK: [test template] *****
changed: [172.17.42.102]
changed: [172.17.42.101]
changed: [172.17.42.103]
changed: [172.17.42.1]

PLAY RECAP *****
172.17.42.1      : ok=2    changed=1    unreachable=0    failed=0
172.17.42.101   : ok=2    changed=1    unreachable=0    failed=0
172.17.42.102   : ok=2    changed=1    unreachable=0    failed=0
172.17.42.103   : ok=2    changed=1    unreachable=0    failed=0
```

我们再挑选一台机器看看它生产后的文件内容，如下所示:

```
[root@703bb6924049 /]# cat /tmp/cpis
=====
```

```
groups info
{'ungrouped': ['172.17.42.1'], 'all': ['172.17.42.1', '172.17.42.101',
    '172.17.42.102', '172.17.42.103'], 'ansible': ['172.17.42.101',
    '172.17.42.102', '172.17.42.103'], 'docker': ['172.17.42.101',
    '172.17.42.102', '172.17.42.103']}
```

```
=====
```

```
=====docker groups info =====
```

```
=172.17.42.101 eth0 IP is 172.17.0.5
```

```
+172.17.42.101 groups is ['ansible', 'docker']
```

```
_ 172.17.42.101 short is 172
```

```
*['172.17.42.1', '172.17.42.101', '172.17.42.102', '172.17.42.103']
```

```
@/root/inventory
```

```
=172.17.42.102 eth0 IP is 172.17.0.4
```

```
+172.17.42.102 groups is ['ansible', 'docker']
```

```
_ 172.17.42.102 short is 172
```

```
*['172.17.42.1', '172.17.42.101', '172.17.42.102', '172.17.42.103']
```

```
@/root/inventory
```

```
=172.17.42.103 eth0 IP is 172.17.0.3
```

```
+172.17.42.103 groups is ['ansible', 'docker']
```

```
_ 172.17.42.103 short is 172
```

```
*['172.17.42.1', '172.17.42.101', '172.17.42.102', '172.17.42.103']
```

```
@/root/inventory
```

看到这个结果后，我们再回到前面定义的 Jinja 模板，进行对比后就一目了然了。



需要注意的是，一定要了解每个变量返回的数据结构类型，否则很容易在我们引用变量的时候出现异常或者提示“XXX 没有此属性相关”错误。

## 4.8 本章小结

本章主要围绕 `playbook` 相关的知识进行介绍，因为 `playbook` 是 Ansible 配置管理中最重要组件，所以本书采用专门的章节来讲解 `playbook`。当然 `playbook` 还有其他特性以及相关知识点本书没有进行相应的介绍，读者可以去参考官网文档进一步学习。下一章我们将介绍 Ansible 最佳实践相关的知识，是我们在工作中使用 Ansible 过程中需要注意的地方。

## Ansible 最佳实践

通过前面几章的学习我们已经了解 Ansible 的基本核心功能，这些知识点可以帮助我们完成日常工作中的很多需求，但是如果想在工作环境中开发一个完整的项目还是稍有欠缺，所以本章将介绍如何在大规模生产环境中使用 Ansible。首先我们通过优化 Ansible 执行速度来提高 Ansible 的效率，然后介绍如何规范 Ansible 整个工作目录，最后介绍在多环境下如何更好地使用 Ansible 等内容。

### 5.1 优化 Ansible 速度

有些人说 Ansible 的执行效率比 SaltStack 差，确实，使用默认的 SSH 方式通信，效率远低于 SaltStack 的 zeromq 消息队列。但是通过本章你会发现可以优化 Ansible 的执行速度，可以做到并不比 SaltStack 差。我在本地 Docker 环境下开启 10 个容器的对比实例如下所示：

```
[root@vm10-160-112-18 ~]# time ansible all -m ping -o
172.17.0.9 | success >> {"changed": false, "ping": "pong"}
172.17.0.4 | success >> {"changed": false, "ping": "pong"}
172.17.0.7 | success >> {"changed": false, "ping": "pong"}
172.17.0.5 | success >> {"changed": false, "ping": "pong"}
172.17.0.2 | success >> {"changed": false, "ping": "pong"}
172.17.0.3 | success >> {"changed": false, "ping": "pong"}
```

```

172.17.0.6 | success >> {"changed": false, "ping": "pong"}
172.17.0.8 | success >> {"changed": false, "ping": "pong"}
172.17.0.10 | success >> {"changed": false, "ping": "pong"}
172.17.0.11 | success >> {"changed": false, "ping": "pong"}
real      0m1.099s
user      0m0.613s
sys       0m0.343s
[root@vm10-160-112-18 ~]# time salt \* test.ping --output=no_return
0a862dd45191:
b771903b732f:
7c5df32ad76f:
767d6150919b:
d5b7a49f14d4:
3b9804cf8605:
5e1a43dfa76a:
e324a3ed0b58:
ff9f0625e3c1:
fac12b82e143:

real      0m1.589s
user      0m1.162s
sys       0m0.051s

```

下面我们就开始介绍如何优化 Ansible 的速度。

## 1. 开启 SSH 长连接

我们知道 Ansible 模式是使用 SSH 和远端主机进行通信，所以 Ansible 对 SSH 的依赖性非常强，这节我们就从 SSH 入手来优化 Ansible。在 OpenSSH 5.6 版本以后 SSH 就支持了 Multiplexing，关于这个特性我们可以参考文章：<https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Multiplexing>。所以如果 Ansible 中控机的 SSH -V 版本高于 5.6 时，我们可以直接在 ansible.cfg 文件中设置 SSH 长连接即可。设置参数如下：

```
sh_args = -o ControlMaster=auto -o ControlPersist=5d
```

ControlPersist=5d 这个参数是设置整个长连接保持时间这里设置为 5 天。如果开启后，通过 SSH 连接过的设备都会在当前 ansible/cp/ 目录下生成一个 socket 文件。也可以通过 netstat 命令查看，会发现有一个 ESTABLISHED 状态的连接一直与远端设备进行着 TCP 连接：

```

tcp 0 0 172.17.42.1:27607 172.17.0.11:22 ESTABLISHED
tcp 0 0 172.17.42.1:40503 172.17.0.4:22 ESTABLISHED
tcp 0 0 172.17.42.1:6911 172.17.0.8:22 ESTABLISHED
tcp 0 0 172.17.42.1:56064 172.17.0.7:22 ESTABLISHED
tcp 0 0 172.17.42.1:54425 172.17.0.6:22 ESTABLISHED
tcp 0 0 172.17.42.1:59930 172.17.0.9:22 ESTABLISHED
tcp 0 0 172.17.42.1:64755 172.17.0.3:22 ESTABLISHED
tcp 0 0 172.17.42.1:48651 172.17.0.5:22 ESTABLISHED
tcp 0 0 172.17.42.1:9974 172.17.0.2:22 ESTABLISHED
tcp 0 0 172.17.42.1:18154 172.17.0.10:22 ESTABLISHED

```

如果我们的中控机 SSH -V 版本低于 5.6 时，则需要升级到 5.6 版本后才能启用 SSH 的 Multiplexing 特性。对于中控机系统是 centos 系统时可以直接使用 yum 源升级版本。下面是 centos 系统的一个 repo 文件，然后运行 yum update openssh-clients 即可完成升级操作。

```

# cat /etc/yum.repos.d/openssh.repo
[CentALT]
name=CentALT Packages for Enterprise Linux 6 - $basearch
baseurl=http://mirror.neu.edu.cn/CentALT/6/$basearch/
enabled=1
gpgcheck=0

```

如果中控机不方便使用 yum 去升级 openssh-clients 时，也可以直接从其他机器复制一个高版本的 SSH2 进制文件，然后软链接到本地的 /usr/bin/ssh 即可完成升级操作。

## 2. 开启 pipelining

pipelining 也是 OpenSSH 的一个特性，我们在第 1 章的时候学习了 Ansible 的整个执行流程，其中有一个流程就是把生成好的本地 Python 脚本 PUT 到远端服务器。如果开启了 pipelining，这个过程将在 SSH 的会话中进行，这样可以大大提高整个执行效率。当然开启 pipelining，需要被控制机 /etc/sudoers 文件编辑当前 Ansible SSH 用户的配置为 requiretty。否则在执行 Ansible 的时候会提示 sudo: sorry, you must have a tty to run sudo。下面我们通过一个示例展示整个过程。首先在 ansible.cfg 配置文件中设置 pipelining 为 True。

```

pipelining = True

```

再来看开启了 pipelining 之后整个 Ansible 执行流程有什么变化。

开启 pipelining 之前的流程如下：

```
[root@vm10-160-112-18 ~]# ansible 172.17.0.2 -a 'w' -vvvv
ESTABLISH CONNECTION FOR USER: root
REMOTE_MODULE command w #USE_SHELL
EXEC sshpass -d6 ssh -C -tt -vvv -o ControlMaster=auto -o
ControlPersist=5d -o ControlPath="/root/.ansible/cp/ansible-ssh-
%h-%p-%r" -o StrictHostKeyChecking=no -o GSSAPIAuthentication=no
-o PubkeyAuthentication=no -o ConnectTimeout=10 172.17.0.2
/bin/sh -c 'mkdir -p $HOME/.ansible/tmp/ansible-
tmp-1435415232.17-6153092968405 && echo $HOME/.ansible/tmp/
ansible-tmp-1435415232.17-6153092968405'
PUT /tmp/tmpn5vV9X TO /root/.ansible/tmp/ansible-
tmp-1435415232.17-6153092968405/command
EXEC sshpass -d6 ssh -C -tt -vvv -o ControlMaster=auto -o
ControlPersist=5d -o ControlPath="/root/.ansible/cp/ansible-ssh-
%h-%p-%r" -o StrictHostKeyChecking=no -o GSSAPIAuthentication=no
-o PubkeyAuthentication=no -o ConnectTimeout=10 172.17.0.2 /bin/sh
-c 'LANG=C LC_CTYPE=C /usr/bin/python /root/.ansible/tmp/ansible-
tmp-1435415232.17-6153092968405/command; rm -rf /root/.ansible/
tmp/ansible-tmp-1435415232.17-6153092968405/ >/dev/null 2>&1'
172.17.0.2 | success | rc=0 >>
14:27:12 up 12 days, 5:05, 1 user, load average: 1.00, 1.01, 1.05
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root pts/0 172.17.42.1 14:27 0.00s 0.03s 0.00s /bin/sh -c LANG
```

开启 pipelining 之后的流程如下：

```
[root@vm10-160-112-18 ~]# ansible 172.17.0.2 -a 'w' -vvvv
ESTABLISH CONNECTION FOR USER: root
REMOTE_MODULE command w #USE_SHELL
EXEC sshpass -d6 ssh -C -vvv -o ControlMaster=auto -o
ControlPersist=5d -o ControlPath="/root/.ansible/cp/ansible-ssh-
%h-%p-%r" -o StrictHostKeyChecking=no -o GSSAPIAuthentication=no
-o PubkeyAuthentication=no -o ConnectTimeout=10 172.17.0.2 /
bin/sh -c 'LANG=C LC_CTYPE=C /usr/bin/python'
172.17.0.2 | success | rc=0 >>
14:27:21 up 12 days, 5:05, 0 users, load average: 1.00, 1.01, 1.05
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
```

我们可以看到开启了 pipelining 之后整个流程少了一个 PUT 脚本去远端服务器的流程，第一步就是调用 sshpass 执行脚本。



### 3. 开启 accelerate 模式

Ansible 还有一个 accelerate 模式，这和前面 SSH 的 Multiplexing 有点类似，因为都依赖 Ansible 中控机跟远端机器有一个长连接。但是 accelerate 是使用 Python 程序在远端机器上运行一个守护进程，然后 Ansible 会通过这个守护进程监听的端口进行通信。开启 accelerate 模式很简单，只要在 playbook 中配置 `accelerate: true` 即可。但是需要注意，如果开启 accelerate 模式，则需要在 Ansible 中控机与远端机器都安装 `python-keyczar` 软件包。下面是在 `ansible.cfg` 文件中定义一些 accelerate 参数，例如远端机器监听端口以及一些 timeout 设置。当然这些参数也可以在写 playbook 的时候再定义：

```
[accelerate]
accelerate_port = 5099
accelerate_timeout = 30
accelerate_connect_timeout = 5.0
```

关于 accelerate 模式的实现过程，建议阅读源码文件 `ansible/modules/core/utilities/helper/accelerate.py` 和 `ansible/runner/connection_plugins/accelerate.py`。如果采用了 accelerate 模式，整个 Ansible 流程将变得不一样了。读者可以使用 `debug` 参数 `-vvvv` 查看整个执行过程。

### 4. 设置 facts 缓存

如果细心的话，就会发现在执行 playbook 的时候，默认第一个 task 都是 GATHERING FACTS，这个过程就是 Ansible 收集每台主机的 facts 信息。方便我们在 playbook 中直接引用 facts 里的信息。当然如果你的 playbook 中不需要 facts 信息，可以在 playbook 中设置 `gather_facts: False` 来提高 playbook 效率。但是如果我们既想在每次执行 playbook 的时候都能收集 facts，又想加速这个收集过程，那么就需要配置 facts 缓存了。目前 Ansible 支持使用 json 文件存储 facts 信息。下面我们首先通过示例来了了解如何使用 json 文件存储 facts 信息：

```
gathering = smart
fact_caching_timeout = 86400
fact_caching = jsonfile
fact_caching_connection = /dev/shm/ansible_fact_cache
```

这里设置 facts 过期时间为 86400 秒（会根据文件的最后修改时间来确定 facts 信息是否过期）。json 文件存放在 `/dev/shm/ansible_fact_cache` 下，下面我们执行一下 playbook：

```
[root@vm10-160-112-18 ~]# ansible-playbook site.yaml -l 172.17.0.2
PLAY [all] *****
GATHERING FACTS *****
ok: [172.17.0.2]
TASK: [test] *****
changed: [172.17.0.2]
PLAY RECAP *****
172.17.0.2 : ok=2 changed=1 unreachable=0 failed=0
```

我们再执行 playbook 的时候就没有 facts 收集这个过程了。直接从 json 文件中读取 facts 缓存信息：

```
[root@vm10-160-112-18 ~]# ansible-playbook site.yaml -l 172.17.0.2
PLAY [all] *****
TASK: [test] *****
changed: [172.17.0.2]
PLAY RECAP *****
172.17.0.2 : ok=1 changed=1 unreachable=0 failed=0
```

再来查看 facts 的 json 存放文件：

```
[root@vm10-160-112-18 ~]# ls -l /dev/shm/ansible_fact_cache/*
-rw-r--r-- 1 root root 6700 6月 27 23:52 /dev/shm/ansible_fact_cache/172.17.0.2
```

文件内容就是一个 json 文件。文件名称是 inventory hostname。可以通过 `cat /dev/shm/ansible_fact_cache/172.17.0.2 | python -m json.tool` 查看文件内容。下面我们再来通过本机的 Redis 存储 facts 信息。目前 facts 存储还不支持远端，所以需要在 Ansible 中控机上安装 Redis 服务，然后安装 Redis Python 库。使用 `yum install redis -y` 安装 Redis 服务即可。可以使用 `pip install redis` 安装 Redis Python 库。首先配置 `ansible.cfg` 文件如下代码所示：

```
gathering = smart
fact_caching_timeout = 86400
fact_caching = redis
```

然后继续运行上面测试的 playbook。可以再开一个终端查看本机的 Redis 信息。

Alt text

最后可以连接到 Redis 上查看 KEY 信息:

```
[root@vm10-160-112-18 ~]# redis-cli
127.0.0.1:6379> KEYS *
1) "ansible_cache_keys"
2) "ansible_facts172.17.0.9"
3) "ansible_facts172.17.0.11"
4) "ansible_facts172.17.0.7"
5) "ansible_facts172.17.0.3"
6) "ansible_facts172.17.0.2"
7) "ansible_facts172.17.0.5"
8) "ansible_facts172.17.0.6"
9) "ansible_facts172.17.0.4"
10) "ansible_facts172.17.0.10"
11) "ansible_facts172.17.0.8"
127.0.0.1:6379>
```

Ansible 的 facts 存储还支持 memcached 存储, 配置方法也很简单, 在 memcached 服务运行后配置 ansible.cfg 即可。这里就不再介绍了。需要注意的是, 要安装 python-memcached 依赖库:

```
gathering = smart
fact_caching_timeout = 86400
fact_caching = memcached
```

## 5.2 目录结构

在日常使用 Ansible 做配置管理工作中, 经常会遇到很多 role 和 playbook 文件和目录。还有一些自定义的模块, 等等。其实官方没有规定一个任务是通过 playbook 文件直接使用还是使用 role 形式, 在前面的章节我们也介绍了如果使用 role 形式封装你的 playbook, 并且任务依赖文件或者依赖其他的任务时, 建议还是以 role 形式存在, 这样方便日后管理和维护。如果只是一个简单的独立任务, 只使用 playbook 文件即可, 这样方便我们在其他地方进行引用。下面介绍官网最佳实践中推举使用的 Ansible 工作目录的结构。统一工作目录如下:

```
production # production 环境的 inventory 文件
```

```

stage                                # stage 环境的 inventory 文件

group_vars/
    group1                          # group1 定义的变量文件
    group2                          # group2 定义的变量文件
host_vars/
    hostname1                      # hostname1 定义的变量文件
    hostname2                      # hostname2 定义的变量文件

library/                            # 自定义模块存放目录
filter_plugins/                    # 自定义 filter 插件存放目录

site.yml                           # playbook 统一入口文件
webservers.yml                     # 特殊任务 playbook 文件

roles/                             # role 存放目录
    common/                        # common 角色 目录
        tasks/                    #
            main.yml              # common 角色 task 入口文件
        handlers/                 #
            main.yml              # common 角色 handlers 入口文件
        templates/                #
            ntp.conf.j2           # common 角色 templates 文件
        files/                   #
            bar.txt               # common 角色 files 资源文件
            foo.sh                # common 角色 files 资源文件
        vars/                    #
            main.yml              # common 角色 变量定义文件
        defaults/                 #
            main.yml              # common 角色 变量定义文件（优先级低）
        meta/                     #
            main.yml              # common 角色依赖文件

    webtier/                       # webtier 角色目录（与 common 角色目录同级）
    monitoring/                   # monitoring 角色目录（与 common 角色目录同级）
    fooapp/                       # fooapp 角色目录（与 common 角色目录同级）

```

使用这种方式规范 Ansible 目录结构对日后维护与管理有很大帮助，当然这个规范方式不一定是最好的，也不一定合适每个人，大家可以根据自己的实际环境进行相应调整。

## 5.3 定义多环境

在实际的工作环境中可能会遇到不同环境的机器，比如笔者公司的流程是，需要部署一套 feature 环境给开发者去做功能开发，等功能开发完成后，又需要给 QA 部门部署一套 stage 环境，最后部署到生产环境时又需要针对 production 环境进行一次部署。其实在整个部署环节里有很多重复性的工作，变化的可能就是目标机器以及一些特殊的配置。那么如何在实际工作中去考虑不同环境中的部署问题呢，下面给大家介绍一下对应的操作。

在笔者公司有一个 CMDB 系统里面存放着所有环境的机器，我们采用脚本的方式定期去 CMDB 里拉取每个环境每个业务角色的机器，然后生成三个 Inventory 文件 (production、stage 和 feature)，最后采用多 Inventory 方式进行引用。这些文件里面的内容是根据每个业务进行分组的，例如 production 环境的某种业务角色的组名称叫作 P@业务名，也就是说 P@mongodb 代表 production 环境的 mongodb 主机组。这是第一种方式，即通过 Inventory 方式去区分多环境下的主机或者主机组信息。接下来的工作就是怎么能够尽量编写重复的 playbook 或者 role 呢。这个时候我们可以根据不同环境配置管理中的配置方法存在哪些异同进行整合。如果所有环境中都是一样的，我们就可以直接写一个 task，如果不同环境需要调用不同的 playbook 或者 task，可以通过 when 方式去判断当前的主机信息存在哪个环境中，然后进行引用。

## 5.4 灰度发布与检测

在实际配置部署过程中为了保证整个配置部署的效率，我们一定要对自己编写的 playbook 或者 role 进行相应的测试后，才能应用到生产环境中，下面就介绍怎样通过检测保证配置部署的效率。

### 1. 语法检测

在编写完 playbook 或者 role 之后一定要养成进行语法检测的习惯，如果编写的 playbook 或者 role 都存在语法问题，在真正实际部署过程中面对很多机器的时候我们会发现调试是一个很麻烦的问题。语法检测也很简单，直接使用 ansible-playbook 命令的 --syntax-check 参数即可。



## 2. 灰度发布

如果语法检测通过之后，我们需要对相关任务进行预运行，灰度发布是什么意思呢？就是先挑选一台机器进行测试，只有进行测试之后我们才知道整个配置流程是否达到我们想要结果。进行预运行时，我们只需要把一个或者多个 task 使用 `delegate_to` 参数指定到一台设备上进行测试。如果测试通过后，再进行接下来的工作。

## 3. 是否达到预想

如果检测通过，我们接下来需要做的就是针对所有机器进行部署。特别是配置文件变更时，我们又没有信心保证所有的配置是否正确生成，此时我们就可以在运行 playbook 的时候使用 `--check` 和 `--diff` 参数去对比生成后的文件是否为我们所需的文件。

## 5.5 统一管理

在运维工作协作的时代，我们经常需要和别人一起去完成配置管理工作，如果 Ansible 作为统一配置管理工具使用，会有很多人参与到 Ansible 的 playbook 和 role 的编写。虽然前面已经介绍了如何去规范整个工作目录。但是每个人的工作方式不同，所以笔者建议把所有的 Ansible 工作目录使用 gitlab 或者 GitHub 私有仓库的形式进行管理。这样使得我们的远程工作协作变得很方便，每个人都可以进行自己的配置管理工作且互不影响。这里还需要做的就是规定一个 git 相关的规范，能保证每个 git 仓库按照规范去使用 git 进行相关的操作即可。

## 5.6 使用 ansible-shell 交互命令行

ansible-shell 是 GitHub 上的一个开源项目，它通过交互式的模式把所有 Ansible 的 Ad-Hoc 命令都引入了，它的底层是引用 Ansible 的 runnerAPI 实现的。整个项目其实就是一个 Python 脚本。ansible-shell 目前能使用所有的 Ad-Hoc 命令而且还支持 Tab 键补齐方式。所以引入这个工具从很大程度上提高了我们使用 Ad-Hoc 命令的频率。目前 ansible-shell 只支持 Ad-Hoc 命令不支持 playbook。下面通过示例对 ansible-shell 进行介绍。首先来安装这个命令，安装方式如下：

```
git clone https://github.com/dominis/ansible-shell.git
cd ansible-shell/
```

```
python setup.py install
```

安装后会在 /usr/bin/ 生成一个 `ansible-shell` 命令，这个时候我们就可以通过运行 `ansible-shell` 命令进入 `ansible-shell`，默认不跟任何参数，但 `ansible-shell` 会引入 `ansible.cfg` 里面的参数。当然也可以在运行 `ansible-shell` 命令时指定参数，比如指定 `inventory`、`ansible-shell -i hosts`、`sudo` 模式 `ansible-shell -s`，还可以指定用户信息，等等。当然还可以在 `ansible.cfg` 文件中添加对 `ansible-shell` 的参数定义，如下所示：

```
[ansible-shell]
cwd= 172.17.0.9
forks= 8
```

下面我们就来体验下 `ansible-shell`：

```
[root@vm10-160-112-18 ~]# ansible-shell
Welcome to the ansible-shell.
Type help or ? to list commands.

root@172.17.0.9 (1) [f:8]$ !hostname
172.17.0.9 | success | rc=0 >>
d5b7a49f14d4

root@172.17.0.9 (1) [f:8]$ copy src=/root/site.yaml dest=/tmp/site.yaml
172.17.0.9 | success >> {
  "changed": true,
  "checksum": "fbfe59ac275ecc427248686b4a3a9ef17f161e5e",
  "dest": "/tmp/site.yaml",
  "gid": 0,
  "group": "root",
  "md5sum": "a0d6c3d2b575a9f9deadaf55dcdc9605",
  "mode": "0644",
  "owner": "root",
  "size": 176,
  "src": "/root/.ansible/tmp/ansible-tmp-1436083261.47-131634914938882/
    source",
  "state": "file",
  "uid": 0
}
```

其他的模块大家可以自己去体验。这里所有的模块名都支持 TAB。默认 `ansible-shell` 有几个内置命令，如下所示：

- `cd` 切换 Inventory 对象（支持正则）
- `list` 显示当前目录下的主机和主机组列表。
- `forks` 临时设置并发数。
- `become` 设置 `become` 模式，例如 `su` 或 `sudo`。
- `!` 强制调用 `shell` 模块。

目前模块的参数不支持 TAB，但是可以输入 `help` 模块名来查看模块参数帮助信息，具体信息如下所示：

```
root@172.17.0.9 (1) [f:8]$ help shell
Execute commands in nodes.
Parameters:
    warn if command warnings are on in ansible.cfg, do not warn about
    this particular line if set to no/false.
    creates a filename, when it already exists, this step will B(not)
    be run.
    executable change the shell used to execute the command. Should be
    an absolute path to the executable.
    chdir cd into this directory before running the command
    removes a filename, when it does not exist, this step will B(not)
    be run.
    free_form The shell module takes a free form command to run, as a
    string. There's not an actual option named "free form". See
    the examples!
```

## 5.7 本章小结

本章主要介绍如何把 Ansible 更好地应用到实际工作中，包括一些日常优化以及如何更好地规范和使用 Ansible，等等。在实际工作中，我们不仅仅把 Ansible 当作一个命令使用，在使用 Ansible 维护和部署一些企业架构后，规模会变得越来越大大且业务也会越来越复杂，所以对 Ansible 的日常维护就变得越来越重要了，所以规范化整个流程也变得至关重要。下一章我们会介绍如何去扩展一些 Ansible 的常用组件，因为每个软件都无法满足所有用户的需求，为了使 Ansible 能满足更加复杂的应用场景，必须对它进行扩展。目前 Ansible 软件本身的扩展性也非常强大。读者不需要去阅读大量的底层源码就可以很简单地对 Ansible 进行扩展。当然如果读者有能力，可以好好分析 Ansible 的源码，会有意想不到的惊喜。

## 扩展 Ansible 组件

通过前几章的学习我们已经了解了 Ansible 的大部分功能以及日常的使用，本章我们将继续学习如何扩展 Ansible 使它能满足我们更加复杂的应用场景。本章将围绕 Ansible 常用的一些组件进行扩展以及自定义，比如我们可以编写自己的 facts 采集方式，还可以编写自己的 Ansible 模块，等等。本章将讲解扩展 facts 和编写日常模块以及一些功能插件的实现。当然 Ansible 的其他插件也都支持扩展，比如 action 插件、connection 插件、vars 插件，但是限于篇幅，本书对以上三种插件不再进行介绍。

### 6.1 扩展 facts

在第3章我们已经了解 facts 相关的知识了，也讲解了通过 `facter` 和 `ohai` 去扩展 facts 信息收集，等等。这节我们将继续研究扩展 facts 信息。在日常不同业务的配置部署过程中肯定会遇到不同业务、不同机器，要使用不同的配置信息。我们完全可以使用 facts 信息来区分这些设备。当然 facts 默认收集的信息大多数参数都是跟机器属性相关的，缺少业务角色相关的属性。我们一般通过主机名或者静态文件描述的方式来表明某台机器的身份。甚至还会通过联合 CMDB 系统来确定机器的业务角色，等等。现在我们将介绍两种采集 facts 信息的方式。

## 1. 定义 facts.d 信息

有时候为了区分一台机器的业务角色或者属性，经常会在设备初始化之后生产一个静态的文件，这个文件一般会包含这台设备的业务相关信息，相当于这台设备的业务名片。Ansible 还支持读取被控制机器文件的方式来当作 facts 信息的数据来源。如果大家看过 Ansible 的 facts 信息收集 setup 模块的源文件，就会非常熟悉整个 facts 收集过程。其实在 Ansible 的 setup 模块采集过程中会检测被控制键的 fact\_path 目录下所有以 fact 结尾的文件，然后读取文件内容当作 facts 信息收集。下面来看 module\_utils/facts.py 文件的 Facts 类下的 get\_local\_facts 函数：

```
def get_local_facts(self):
    fact_path = module.params.get('fact_path', None)
    if not fact_path or not os.path.exists(fact_path):
        return
    local = {}
    for fn in sorted(glob.glob(fact_path + '/*.fact')):
        # where it will sit under local facts
        fact_base = os.path.basename(fn).replace('.fact', '')
        if stat.S_IXUSR & os.stat(fn)[stat.ST_MODE]:
            # run it
            # try to read it as json first
            # if that fails read it with ConfigParser
            # if that fails, skip it
            rc, out, err = module.run_command(fn)
        else:
            out = get_file_content(fn, default='')

        # load raw json
        fact = 'loading %s' % fact_base
        try:
            fact = json.loads(out)
        except ValueError, e:
            # load raw ini
            cp = ConfigParser.ConfigParser()
            try:
                cp.readfp(StringIO.StringIO(out))
            except ConfigParser.Error, e:
                fact="error loading fact - please check content"
            else:
```



```

        fact = {}
        #print cp.sections()
        for sect in cp.sections():
            if sect not in fact:
                fact[sect] = {}
            for opt in cp.options(sect):
                val = cp.get(sect, opt)
                fact[sect][opt]=val

        local[fact_base] = fact
    if not local:
        return
    self.facts['local'] = local

```

fact\_path 路径是在 moudles/core/system/setup.py 模块文件的 main 函数里面指定的，默认路径是 /etc/ansible/facts.d。当然 setup 模块还支持指定 fact\_path 路径。在了解整个 facts 信息收集流程之后，我们就可以很简单地在 fact\_path 目录下生成想要的文件，然后当作 facts 信息来收集了。glob.glob (fact\_path + '/\*.fact') 所有的静态文件必须是以 fact 结尾的，目前文件内容格式只支持 JSON 和 INI 格式。如果是 JSON 格式它会直接使用 json.load 方式进行解析，如果是 INI 格式它会使用 ConfigParser 模块进行解析。下面我们通过一个示例来了解整个流程：

```

[root@Master facts.d]# pwd
/etc/ansible/facts.d
[root@Master facts.d]# cat ansible.fact
{ "name": "shencan" , "list": ["three", "one", "two"], "Dict": { "A": "B" } }

```

在这里为了大家能了解各种数据结构，我们首先定义了三种数据类型，然后这个文件使用 JSON 格式的，最后测试这个 facts 信息是否能被正确解析：

```

[root@Master facts.d]# ansible -i /root/hosts 192.168.1.116 -m setup
-a "filter=ansible_local"
192.168.1.116 | success >> {
    "ansible_facts": {
        "ansible_local": {
            "ansible": {
                "Dict": {
                    "A": "B"
                },

```

```

"list": [
    "three",
    "one",
    "two"
],
"name": "shencan"
}
},
"changed": false
}

```

## 2. 编写 facts 模块

上面介绍的方式灵活性不是很强，因为需要每台机器上都存在一个那样的文件。这节我们通过编写模块的方式去收集 facts 信息，就是相当于自己写一个功能跟 setup 模块类似的模块。如果只是编写 facts 信息采集模块，编写流程很简单，只需要按照编写 modules 的要求编写即可，唯一要求是在最后需要把所有的 facts 信息（JSON 格式）存储到 `ansible_factskey` 下。通过如下示例来了解模块编写原理：

```

[root@Master ~]# cat library/info.py
#!/usr/bin/python
DOCUMENTATION = """
---
module: info
short_description: This modules is extend facts modules
description:
    - This modules is extend facts modules
version_added: "1.1"
options:
    enable:
    description:
        - enable extend facts
    required: true
    default: null
"""

EXAMPLES = '''
- info: enable=yes
'''

import json

```

```

import shlex
import sys
args_file = sys.argv[1]
args_data=file(args_file).read()
arguments=shlex.split(args_data)
for arg in arguments:
    if "=" in arg:
        (key,value) = arg.split("=")
        if key == "enable" and value == "yes":
            data={}
            data['key'] = 'value'
            data['list'] = ['one','two','three']
            data['dict'] = {'A':"a"}
            print json.dumps({"ansible_facts": data},indent=4)
        else:
            print "info modules usage error"
    else:
        print "info modules need one parameter"

```

这是一个非常简单的 Python 脚本，也是一个非常简单的 Ansible 模块。前面 DOCUMENTATION 和 EXAMPLES 是为了 ansible-doc 能查看这个模块的简介和例子。我们来测试一下这个模块：

```

[root@Master ~]# ansible all -M library/ -m info -a 'enable=yes' -o
172.17.42.104 | success >> {"ansible_facts": {"dict": {"A": "a"}, "key":
    "value", "list": ["one", "two", "three"]}}

172.17.42.103 | success >> {"ansible_facts": {"dict": {"A": "a"}, "key":
    "value", "list": ["one", "two", "three"]}}

172.17.42.102 | success >> {"ansible_facts": {"dict": {"A": "a"}, "key":
    "value", "list": ["one", "two", "three"]}}

172.17.42.101 | success >> {"ansible_facts": {"dict": {"A": "a"}, "key":
    "value", "list": ["one", "two", "three"]}}

```

因为这个模块不在 Ansible 默认模块路径下，所以执行 Ad-hoc 命令时需要指定模块路径。如果模块在当前目录的 library/ 下就无需指定模块路径。当然可以把模块路径追加到 ANSIBLE\_LIBRARY 这个环境变量下，然后通过 playbook 的 template 去引用这些 facts 信息。首先介绍 Jinja2 模板文件和 playbook 文件：

```
key is {{ key }}
list is {% for i in list %} {{ i }} {% endfor %}
```

```
dict['A'] is {{ dict['A'] }}
```

playbook 文件如下：

```
---
- hosts: all
  tasks:
    - name: test info facts module
      info: enable=yes
    - name: debug info facts
      template: src=info.j2 dest=/tmp/cpis
```

最后我们来运行这个 playbook：

```
[root@Master ~]# ansible-playbook -i hosts info.yaml -l 192.168.1.116
```

```
PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.116]

TASK: [test info facts module] *****
ok: [192.168.1.116]

TASK: [debug info facts] *****
ok: [192.168.1.116]

PLAY RECAP *****
192.168.1.116      : ok=3    changed=0    unreachable=0    failed=0
```

查看渲染后的文件：

```
[root@Master ~]# ansible -i hosts 192.168.1.116 -a 'cat /tmp/cpis'
192.168.1.116 | success | rc=0 >>
```

```
key is value
list is one two three
dict['A'] is a
```

## 6.2 扩展模块

在上一节编写 facts 模块的时候，我们已经编写了一个 Ansible 模块。这节我们将继续深入讲解如何编写模块。关于编写模块所使用的语言，目前官方支持用任何语言编写模块。当然本节也会介绍运维经常使用的两种用来编写模块的语言。

### 1. 使用 shell 编写模块

如果我们使用 shell 去编写 Ansible 的模块，这个流程是非常简单的。我们平常使用模块的时候一般是指定模块名，然后通过传入 key/value 键值对的参数，最后在模块执行完成后会有模块执行状态以及相关结果返回。下面我们通过一个示例来了解整个数据流程：

```
[root@Master ~]# cat library/docker_sh
#!/bin/bash
set -e
source $1 $2 $3
IMAGE=$image
NAME=$name
TAG=$tag
if [ ! -z "$IMAGE" ] && [ ! -z "$NAME" ] && [ ! -z "$TAG" ]; then
    id=$(/usr/bin/docker run --name $NAME -d ${IMAGE}:${TAG})
    if [ ! -z "$id" ]; then
        CHANGED="True"
        echo {"containerid":\\"$id\\"}
        exit 0
    fi
else
    echo { \"msg\": \"run docker container error\" }
    exit 0
fi
```

这里使用 shell 完成了一个 docker\_sh 模块，这个模块会根据传入的 image tag name 去运行 Docker 容器。最后把容器 id 打印出来。测试如下：

```
[root@Master ~]# ansible -i hosts 192.168.1.116 -M library/ -m docker_
sh -a 'image=shencan/fuck name=ansible tag=v5' -c local
192.168.1.116 | success >> {
    "containerid": "1d2ea83c8c6c8e9337afe1ccc9dclb422b1e13dbabeba8f16962
    e659ce51b84b6"
}
```



然后查看运行的 docker 容器：

```
[root@Master ~]# docker ps -a|grep '1d2ea83c8c6' -B 1
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1d2ea83c8c6c shencan/fuck:v5 "/usr/bin/supervisor 3 minutes ago Up 3
minutes 22/tcp ansible
```

使用 shell 编写模块需要注意最后脚本的输出，因为 Ansible 最后会使用 json.load 把结果格式化，所以一定需要 echo 的结果能被 Ansible 解析。关于编写模块在 playbook 中的应用，可以参照编写 facts 模块一节的内容。我们可以看到引用模块方式与日常所写的 playbook 没有任何区别，仅需要注意编写的模块路径。

## 2. 使用 Python 编写模块

虽然使用 shell 可以完成某些 module 编写，但有些 Python 数据结构在 shell 中表现形式不一样。本节我们使用 Python 语言去编写模块。这样我们就可以和 Ansible 紧密结合了，甚至可以直接导入 Ansible 内部的模块。这里选择了笔者在 yottaa 工作时，编写的一个 openstack 分配 floating 的示例。当时我们使用 Ansible 去管理 AWS 和 OpenStack，发现官网对一些功能模块不支持，所以我们就使用 OpenStack 和 AWS 的一些 API 进行封装，做成 Ansible 的模块。事实上使用 Python 编写模块也是最简单的，现在我们一起了解此示例：

```
[root@Master ~]# cat library/floating
#!/usr/bin/env python
# -*- coding: utf-8 -*-
DOCUMENTATION = """
---
module: floating
short_description: This modules pull floating ipaddress from openstack
description:
    - This modules pull floating ipaddress from openstack
version_added: "1.1"
options:
    usernmae:
        description:
            - openstack auth username
        required: true
```

```

        default: admin
    password:
        description:
            - openstack auth password
        required: true
        default: password
    tenant:
        description:
            - openstack tenant information
        required: true
        default: admin
    authurl:
        description:
            - openstack authurl
        required: true
        default: http://10.8.6.2:35357/v2.0/
author: Can Shen
requirements: [ "python-novaclient >= 2.20.0" ]
"""
EXAMPLES = '''
- name: pull floating ip address
  local_action: floating
    username={{ login_username }} password={{ login_
        password }}
    tenant={{ login_tenant_name }} authurl={{ auth_url }}
  register: floating_ip
'''

import sys
import json
import os, commands
from ansible.module_utils.basic import *
from novaclient.v1_1 import client

def main():
    module = AnsibleModule(
        argument_spec = dict(
            username = dict(default='admin', type='str'),
            password = dict(default='password', type='str'),
            tenant = dict(default='admin.', type='str'),
            authurl = dict(default='http://10.8.6.2:35357/
                v2.0/', type='str')
        ),

```

```

    supports_check_mode=True
)
USER,PASS,TENANT,AUTH_URL=(module.params['username'],module.
    params['password'],module.params['tenant'],module.
    params['authurl'])
openstack = client.Client(USER, PASS, TENANT, AUTH_URL, service_
    type="compute")
ips=[]
for i in openstack.floating_ips.list():
    if i.fixed_ip is None:
        ips.append(i.ip)
floating_ip=str(ips[0])

if not floating_ip :
    print json.dumps({
        "failed" : True,
        "msg"      : "not have floating_ip "
    })
    sys.exit(1)
print json.dumps({
    "res" : floating_ip,
    "changed" : True
})
sys.exit(0)

main()

```

关于这个脚本我们重点介绍加粗的部分。

这几段的意思是引用 `ansible.module_utils.basic` 的 `AnsibleModule` 对模块传入参数的 `key` 和 `value` 进行默认值定义，并定义这个模块支持哪些参数和每个参数的数据类型是什么，等等。`supports_check_mode=True` 是开启模块的 `check` 支持。关于模块引用模块传入的值，`module.params['username']` 所有参数传入的 `key/value` 都在 `module.params` 下，所以可以很简单地取出某个参数传入的值。下面是如何去使用这个模块的例子：

```

[root@Master library]# cat openstack.yaml
---
- name: new openstatck instances
  hosts: 127.0.0.1
  gather_facts: False

```

```

vars:
  auth_url: http://10.8.6.2:35357/v2.0/
  login_username: admin
  login_tenant_name: admin
  login_password: password
  image_id: c1c28aa4-a892-40a0-b7aa-f692fd2785b1
  keypair_name: Yottaa-Stage-v1307
  private_net: 696bfa85-5e8e-4dea-b41c-e066bfa925ac
connection: local
tasks:
  - nova_compute:
      auth_url: "{{ auth_url }}"
      login_username: "{{ login_username }}"
      login_password: "{{ login_password }}"
      login_tenant_name: "{{ login_tenant_name }}"
      security_groups: default
      state: present
      name: "{{ ro }}-{{ 1000000 | random(1, 1) }}.yottaa.com"
      image_id: "{{ image_id }}"
      key_name: "{{ keypair_name }}"
      flavor_id: "{{ flavor_id }}"
      nics:
        - net-id: "{{ private_net }}"
      register: openstack

  - name: pull floating ip address
    local_action: floating
      username={{ login_username }} password={{ login_password }}
      tenant={{ login_tenant_name }} authurl={{ auth_url }}
    register: floating_ip

  - name: bond floating ip to instance
    local_action: bondip_openstack
      username={{ login_username }} password={{ login_password }}
      tenant={{ login_tenant_name }} authurl={{ auth_url }}
      instance_id={{ openstack.id }} floating_ip={{ floating_ip.res }}

```

这个 playbook 的 `bondip_openstack` 和 `floating` 都是使用 Python 编写的模块，这里与我们直接使用 Ansible 自带模块没有任何区别。

## 6.3 callback 插件

### 1. 了解 callback 插件流程

callback 是 Ansible 的一个回调功能，我们可以在运行 Ansible 的时候调用这个功能。比如希望在执行 playbook 失败后发邮件，或者希望将每次执行 playbook 的结果存到日志或者数据库中。在 callback 插件里面，我们可以很方便地拿到 Ansible 执行状态信息，然后可以定义一个 callback 动作，在 playbook 的某个运行状态下进行调用。下面我们通过官网的一个关于记录 ad-hoc 和 playbook 执行记录写入日志文件的 callback 进行讲解，让大家熟悉整个 callback 的调用流程，达到最终编写适合自己需求的 callback。当然 Ansible 本身也有日志记录功能，在 `ansible.cfg` 中开启 `log_path` 即可，但是 callback 可以记录到更加详细的信息，示例如下，脚本名称为 `log_plays.py`：

```
import os
import time
import json

# NOTE: in Ansible 1.2 or later general logging is available without
# this plugin, just set ANSIBLE_LOG_PATH as an environment variable
# or log_path in the DEFAULTS section of your ansible configuration
# file. This callback is an example of per hosts logging for those
# that want it.

TIME_FORMAT="%b %d %Y %H:%M:%S"
MSG_FORMAT="%(now)s - %(category)s - %(data)s\n\n"

if not os.path.exists("/var/log/ansible/hosts"):
    os.makedirs("/var/log/ansible/hosts")

def log(host, category, data):
    if type(data) == dict:
        if 'verbose_override' in data:
            # avoid logging extraneous data from facts
            data = 'omitted'
        else:
            data = data.copy()
            invocation = data.pop('invocation', None)
            data = json.dumps(data)
            if invocation is not None:
```



```

data = json.dumps(invocation) + " => %s " % data
path = os.path.join("/var/log/ansible/hosts", host)
now = time.strftime(TIME_FORMAT, time.localtime())
fd = open(path, "a")
fd.write(MSG_FORMAT % dict(now=now, category=category, data=data))
fd.close()

```

```

class CallbackModule(object):

```

```

    """
    logs playbook results, per host, in /var/log/ansible/hosts
    """

```

```

    def on_any(self, *args, **kwargs):
        pass

```

```

    def runner_on_failed(self, host, res, ignore_errors=False):
        log(host, 'FAILED', res)

```

```

    def runner_on_ok(self, host, res):
        log(host, 'OK', res)

```

```

    def runner_on_skipped(self, host, item=None):
        log(host, 'SKIPPED', '....')

```

```

    def runner_on_unreachable(self, host, res):
        log(host, 'UNREACHABLE', res)

```

```

    def runner_on_no_hosts(self):
        pass

```

```

    def runner_on_async_poll(self, host, res, jid, clock):
        pass

```

```

    def runner_on_async_ok(self, host, res, jid):
        pass

```

```

    def runner_on_async_failed(self, host, res, jid):
        log(host, 'ASYNC_FAILED', res)

```

```

    def playbook_on_start(self):
        pass

```

```

def playbook_on_notify(self, host, handler):
    pass

def playbook_on_no_hosts_matched(self):
    pass

def playbook_on_no_hosts_remaining(self):
    pass

def playbook_on_task_start(self, name, is_conditional):
    pass

def playbook_on_vars_prompt(self, varname, private=True,
    prompt=None, encrypt=None, confirm=False, salt_size=None,
    salt=None, default=None):
    pass

def playbook_on_setup(self):
    pass

def playbook_on_import_for_host(self, host, imported_file):
    log(host, 'IMPORTED', imported_file)

def playbook_on_not_import_for_host(self, host, missing_file):
    log(host, 'NOTIMPORTED', missing_file)

def playbook_on_play_start(self, name):
    pass

def playbook_on_stats(self, stats):
    pass

```

我们先不去分析这个脚本，而是在调用这个 callback 之后再来分析它。如果想开启 callback 插件，需要把 callback 脚本放到 callback\_plugins 指定的目录下。在 ansible.cfg 目录下可以指定 callback\_plugins 的路径。下面我们就把官网的这个 callback 脚本放到 callback\_plugins 下，然后执行 Ansible playbook 操作：

```

[root@Master ~]# ll /usr/share/ansible_plugins/callback_plugins/log_
plays.py
-rwxr-xr-x 1 root root 2772 6月 14 16:27 /usr/share/ansible_plugins/
callback_plugins/log_plays.py

```

```
[root@Master ~]# ansible all -m shell -a 'uname -r' -o
172.17.42.102 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
172.17.42.101 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
172.17.42.104 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
172.17.42.103 | success | rc=0 | (stdout) 3.10.5-3.el6.x86_64
[root@Master ~]# ansible-playbook info.yaml
```

```
PLAY [all] *****
```

```
GATHERING FACTS *****
```

```
ok: [172.17.42.103]
```

```
ok: [172.17.42.102]
```

```
ok: [172.17.42.104]
```

```
ok: [172.17.42.101]
```

```
TASK: [test info facts module] *****
```

```
ok: [172.17.42.102]
```

```
ok: [172.17.42.104]
```

```
ok: [172.17.42.103]
```

```
ok: [172.17.42.101]
```

```
TASK: [debug info facts] *****
```

```
ok: [172.17.42.101]
```

```
ok: [172.17.42.102]
```

```
ok: [172.17.42.103]
```

```
ok: [172.17.42.104]
```

```
PLAY RECAP *****
```

```
172.17.42.101 : ok=3    changed=0    unreachable=0    failed=0
172.17.42.102 : ok=3    changed=0    unreachable=0    failed=0
172.17.42.103 : ok=3    changed=0    unreachable=0    failed=0
172.17.42.104 : ok=3    changed=0    unreachable=0    failed=0
```

因为 callback 脚本里面定义的日志在 /var/log/ansible 目录下，我们去这个目录下查看：

```
[root@Master ~]# cd /var/log/ansible
[root@Master ansible]# ls
hosts
[root@Master ansible]# tree
.
├── hosts
└── 172.17.42.101
```

```

|—— 172.17.42.102
|—— 172.17.42.103
|—— 172.17.42.104

```

```
1 directory, 4 files
```

这里是每台主机的日志，写入以 `Invenotory` 名字的文件内。下面我们查看一台机器的日志：

```
[root@Master ansible]# cat hosts/172.17.42.101
Jun 14 2015 16:27:34 - OK - omitted
```

```
Jun 14 2015 16:27:34 - OK - {"module_name": "info", "module_args":
    "enable=yes"} => {"ansible_facts": {"dict": {"A": "a"}, "list":
    ["one", "two", "three"], "key": "value"}}
```

```
Jun 14 2015 16:27:35 - OK - {"module_name": "template", "module_
args": "src=info.j2 dest=/tmp/cpis"} => {"group": "root", "uid":
0, "changed": false, "owner": "root", "state": "file", "gid": 0,
"mode": "0644", "path": "/tmp/cpis", "size": 55}
```

第1条日志是执行模块的，第2条和第3条是执行上面的 `info.yaml` 的日志，里面会记录 `playbook` 中引用的每个 `task` 的执行结果信息。这个时候我们再来看上面的 `callback` 脚本。脚本中定义了一个 `log` 函数，这个函数接受参数传入，然后对传入的 `data` 信息进行解析判定后，将其写入 `path = os.path.join("/var/log/ansible/hosts", host)` 文件：

```
def runner_on_failed(self, host, res, ignore_errors=False):
    log(host, 'FAILED', res)

def runner_on_ok(self, host, res):
    log(host, 'OK', res)

def runner_on_skipped(self, host, item=None):
    log(host, 'SKIPPED', '...')

def runner_on_unreachable(self, host, res):
    log(host, 'UNREACHABLE', res)
```

这里定义了一些 `runner` 的执行状态，只有 `runner` 状态匹配到了才会去引用前面定

义的 log 函数继续日志记录。下面是匹配 playbook 的执行状态，如果不了解每个函数的变量信息，可以直接 print 出来：

```
def playbook_on_import_for_host(self, host, imported_file):
    log(host, 'IMPORTED', imported_file)

def playbook_on_not_import_for_host(self, host, missing_file):
    log(host, 'NOTIMPORTED', missing_file)
```

## 2. 编写 callback 插件

了解了 callback 插件的流程之后，我们就可以根据自己的需求去编写 callback 插件了。下面我们编写一个 callback 插件，把一些 playbook 执行状态信息写入 MySQL 里面。首先我们来看一下 callback 代码：

```
[root@Master callback_plugins]# cat status.py
import MySQLdb
import os
import time
import datetime
TIME_FORMAT='%Y-%m-%d %H:%M:%S'
now = datetime.datetime.now()
def insert(host,res):
    conn=MySQLdb.connect(host='192.168.1.117',user='root',passwd='123456',db='ansible',port=3306)
    cur=conn.cursor()
    sql='insert into status(hosts,result,date) values("%s","%s","%s")'
    %(host,res,now.strftime(TIME_FORMAT))
    cur.execute(sql)
    conn.commit()
    conn.close()

class CallbackModule(object):
    def on_any(self, *args, **kwargs):
        pass
    def runner_on_failed(self, host, res, ignore_errors=False):
        insert(host,res)
    def runner_on_ok(self, host, res):
        insert(host,res)
    def runner_on_unreachable(self, host, res):
        insert(host,res)
```



```

def playbook_on_import_for_host(self, host, imported_file):
    pass
def playbook_on_not_import_for_host(self, host, missing_file):
    pass
def playbook_on_stats(self, stats):
    hosts = stats.processed.keys()
    for i in hosts:
        info = stats.summarize(i)
        if info['failures'] > 0 or info['unreachable'] > 0:
            has_errors = True
        msg = "Hostinfo: %s, ok: %d, failures: %d, unreachable: %d, changed: %d, skipped: %d" % (i, info['ok'], info['failures'], info['unreachable'], info['changed'], info['skipped'])
        print msg

```

这是一个比较简单的 callback 插件，实现将 playbook 结果插入 MySQL。我们来运行 playbook，然后查看下 MySQL 数据：

```

[root@Master callback_plugins]# ansible-playbook -i /root/inventory/
docker /root/info.yaml -l 172.17.42.101:172.17.42.102

```

```

PLAY [all] *****

```

```

GATHERING FACTS *****

```

```

ok: [172.17.42.102]

```

```

ok: [172.17.42.101]

```

```

TASK: [test info facts module] *****

```

```

ok: [172.17.42.101]

```

```

ok: [172.17.42.102]

```

```

TASK: [debug info facts] *****

```

```

ok: [172.17.42.101]

```

```

ok: [172.17.42.102]

```

```

PLAY RECAP *****

```

```

Hostinfo: 172.17.42.102, ok: 3, failures: 0, unreachable: 0, changed: 0,
skipped: 0

```

```

Hostinfo: 172.17.42.101, ok: 3, failures: 0, unreachable: 0, changed: 0,
skipped: 0

```

```

172.17.42.101      : ok=3      changed=0    unreachable=0    failed=0

```

```

172.17.42.102      : ok=3      changed=0    unreachable=0    failed=0

```

我们来查看下 MySQL 的数据。因为这里 result 直接把 res 的数据全部插入对应数据库的表中，当然也可以根据自己的需求只插入所需要的属性。这个表里面的 3 行数据分别代表 setup 模块的返回信息，以及 info.yaml 中两个 task 中的模块返回的数据，如图 6-1 所示。

hosts	result	date
172.17.42.101	{'invocation': {'module_name': 'setup', 'module_args': ''}, 'verbose_override': True, 'changed': False, '2015-06-14 20:56:23	
172.17.42.101	{'invocation': {'module_name': 'u'info', 'module_args': 'u'enable=yes'}, 'ansible_facts': {'dict': {'A': 'a'}}, '2015-06-14 20:56:23	
172.17.42.101	{'group': 'root', 'uid': 0, 'changed': False, 'invocation': {'module_name': 'u'template', 'module_args': 'u'2015-06-14 20:56:23	
172.17.42.102	{'invocation': {'module_name': 'setup', 'module_args': ''}, 'verbose_override': True, 'changed': False, '2015-06-14 20:56:23	
172.17.42.102	{'invocation': {'module_name': 'u'info', 'module_args': 'u'enable=yes'}, 'ansible_facts': {'dict': {'A': 'a'}}, '2015-06-14 20:56:23	
172.17.42.102	{'group': 'root', 'uid': 0, 'changed': False, 'invocation': {'module_name': 'u'template', 'module_args': 'u'2015-06-14 20:56:23	

图 6-1 将数据插入 MySQL

## 6.4 lookup 插件

在第 4 章我们介绍了一些常用的 lookup 插件，并且通过一些简单的例子熟悉了 lookup 的基本运行流程。本节我们将继续介绍 lookup 的相关知识，并且通过一个案例来分享如何去定义自己的 lookup 插件。Ansible 默认的所有 lookup 插件文件在当前 Python 版本的 ansible/runner/lookup\_plugins/ 目录下，有 Python 语言基础的读者建议好好阅读该目录下的 lookup 文件。在本节里我们通过一个从 MySQL 数据库中读取变量值的 lookup 插件示例来熟悉整个 lookup 插件流程后，大家根据实际工作环境的需求，可以很轻松地编写适合自己的 lookup 插件。下面我们就开始编写自己的 lookup 插件。首先搭建一个 MySQL 数据库，存放所有的变量信息，MySQL 的数据库名是 Ansible，表名是 lookup，然后在表里面插入了几条数据，最终我们要使用这些变量的值从这台 MySQL 数据库中读取数据，数据库表内容如图 6-2 所示。

id	keyname	value
1	ansible	cool
2	one	ONE
3	two	TWO

图 6-2 MySQL 数据库中的表

我们在 ansible.cfg 配置文件中 lookup\_plugins 指定的目录下新建一个 mysql.py 的 lookup 查询脚本，内容如下：

```
#!/usr/bin/python
```

```
"""
```

```
Description: This lookup query value from mysql
```

```
Example Usage:
```

```
{{ lookup('mysql', ('192.168.1.117', 'Ansible', 'lookup', 'ansible')) }}
```

```
"""
```

```
from ansible import utils, errors
```

```
HAVE_MYSQL=False
```

```
try:
```

```
    import MySQLdb
```

```
    HAVE_REDIS=True
```

```
except ImportError:
```

```
    pass
```

```
class LookupModule(object):
```

```
    def __init__(self, basedir=None, **kwargs):
```

```
        self.basedir = basedir
```

```
    if HAVE_REDIS == False:
```

```
        raise errors.AnsibleError("Can't LOOKUP(mysql): module  
MySQLdb is not installed")
```

```
    def run(self, terms, inject=None, **kwargs):
```

```
        terms = utils.listify_lookup_plugin_terms(terms, self.basedir,  
            inject)
```

```
        ret = []
```

```
        host,db,table,key=(terms[0],terms[1],terms[2],terms[3])
```

```
        conn=MySQLdb.connect(host=host,user='root',passwd='123456',db=  
            db,port=3306)
```

```
        cur=conn.cursor()
```

```
        sql='select value from %s where keyname = "%s"' % (table,key)
```

```
        cur.execute(sql)
```

```
        result=cur.fetchone()
```

```
        if result[0]:
```

```
            ret.append(result[0])
```

```
            return ret
```

```
        else:
```

```
            return None
```

这个 lookup 脚本的功能比较容易理解，它是通过连接到 lookup 参数的 MySQL 服务器查询相应 key 的值。这里需要注意的是 Ansible 控制机需要安装 MySQLdb Python

库。下面我们通过一个 playbook 来引用我们定义的 MySQL lookup 插件：

```
[root@Master ~]# cat lookup.yaml
---
- hosts: all
  vars:
    value: "{{ lookup('mysql', ('192.168.1.117', 'Ansible', 'lookup',
                                'one')) }}"
  tasks:
    - name: test lookup
      template: src=lookup.j2 dest=/tmp/lookup
```

可以看到传入了 MySQL 的主机 ip、数据库、表名和需要查询的 key。比如这里是去查询 192.168.1.117 上 Ansible 数据库中 lookup 表里面 key 为 one 的值。我们在 lookup.j2 模板里面引用 value 的值即可：

```
{{ value }}
```

最后我们运行这个 playbook，查看渲染后的文件：

```
[root@Master ~]# ansible-playbook /root/lookup.yaml -l 192.168.1.116

PLAY [all] *****

GATHERING FACTS *****
ok: [192.168.1.116]

TASK: [test lookup] *****
changed: [192.168.1.116]

PLAY RECAP *****
192.168.1.116      : ok=2    changed=1    unreachable=0    failed=0

[root@Master ~]# ansible 192.168.1.116 -a 'cat /tmp/lookup'
192.168.1.116 | success | rc=0 >>
ONE
```

## 6.5 Jinja2 filter

在第 4 章讲解 playbook 的时候我们也介绍了 Jinja2 的一些基本使用方法，我们都

知道 Jinja2 的模板定义功能很强大，如果想深入学习 Ansible，建议去深入研究 Jinja2 这个模板语言。我们平常编写 Jinja2 模板的时候经常会使用 Jinja2 的语法以及它强大的 filter。虽然 Jinja2 自带很多 filter，但是在真正的应用场景中经常会遇到自带 filter 解决不了的问题，所以这节我们将讲解如何去定义自己的 Jinja2 filter。

## 1. 了解 Jinja2 filter 的执行过程

Ansible 所有的 Jinja2 filter 由 Ansible 的 filter\_plugins 插件和 Jinja2 自带的 filter 组成。Jinja2 所有自带的 filter 在 Jinja2 当前 Python 版本的 site-packages/jinja2/filter.py 文件内。建议熟悉 Python 语言的读者好好研究这个文件。Ansible 的 filter\_plugins 插件中的 filter，在当前 Python 版本的 site-packages/ansible/runner/filter\_plugins 目录下。下面我们通过 Ansible 的 filter\_plugins 插件中的 fileglob filter 讲解 filter，首先我们来看 Python 版本的 site-packages/ansible/runner/filter\_plugins 目录下 core.py 中的 fileglob 函数：

```
def fileglob(pathname):  
    ''' return list of matched files for glob '''  
    return glob.glob(pathname)
```

这个函数的作用就是使用 glob 模块（脚本开头已经引入了该模块）匹配一个指定目录下的文件，最后返回所有匹配的文件，并在 FilterModule 类的 filters 函数下进行引用：

```
class FilterModule(object):  
    ''' Ansible core jinja2 filters '''  
  
    def filters(self):  
        return {  
            # file glob  
            'fileglob': fileglob,
```

前面的 'fileglob' 是 filter 的名称，后面的 fileglob 是前面定义的 fileglob 函数。下面我们通过一个示例来引用 fileglob 这个 filter。这里是定义的 Jinja2 模板：

```
{% set path = '/root/*.py' %}  
{{ path | fileglob }}
```

然后写一个 playbook 去引用这个 Jinja2 模板：

```
---  
- hosts: all
```



```
tasks:
  - name: test jinja2 filter
    template: src=filter.j2 dest=/tmp/filter
```

最后我们来运行，查看渲染后的文件结果：

```
[root@Master ~]# ansible-playbook filter.yaml -l 172.17.42.103
```

```
PLAY [all] *****
```

```
GATHERING FACTS *****
```

```
ok: [172.17.42.103]
```

```
TASK: [test jinja2 filter] *****
```

```
ok: [172.17.42.103]
```

```
PLAY RECAP *****
```

```
172.17.42.103      : ok=2    changed=0    unreachable=0    failed=0
```

```
[root@Master ~]# ansible 172.17.42.103 -a 'cat /tmp/filter'
```

```
172.17.42.103 | success | rc=0 >>
```

```
['/root/status.py', '/root/hosts.py', '/root/shencan.py', '/root/mysql.py', '/root/filter.py', '/root/inventory-ansible.py']
```

## 2. 定义 Jinja2 filter

通过上一小节我们已经了解了 Ansible 的 filter 执行原理，如果我们想扩展 Ansible 的 filter，可以直接修改上面的 filter Python 脚本，但是不建议直接修改 filter 源码文件，因为这样不利于维护，而且版本升级后还可以被覆盖，所以本节我们通过 Ansible 的 filter plugins 插件扩展方式去定义一个属于自己的 filter。在 ansible.cfg 配置文件里面的 filter\_plugins 值已经指定了 filter\_plugins 的路径，我们只需在这个目录下编写我们的 filter 即可。下面通过很简单的方式定义两个 filter，这里只是展示一下整个 filter 定义的流程，大家可以根据工作中的实际需求去定义即可：

```
[root@Master ~]# cat /usr/share/ansible_plugins/filter_plugins/filter.py
#!/usr/bin/python
from jinja2.filters import environmentfilter
from ansible import errors
import time
```

```
def string(str, seperator=' '):
    return str.split(seperator)

def _time(times):
    return time.mktime(time.strptime(times,'%Y-%m-%d %H:%M:%S'))
```

```
class FilterModule(object):
    ''' Ansible custom Filter'''
    def filters(self):
        return {
            'shencan' : string,
            'shencan1': _time,
        }
```

这里定义了两个 filter，分别是 shencan 和 shencan1，然后 shencan 引用 string 函数，shencan1 引用 \_time 函数，string 函数的作用就是把传入的字符串通过 python str 内置的 split 方法切割成一个 python list。默认的 seperator 是空格，也支持传入 seperator。\_time 函数的作用，例如把 %Y-%m-%d %H:%M:%S 这种格式的时间算成时间戳格式。

下面我们通过编写 Jinja2 模板来引入我们定义的 filter：

```
[root@Master ~]# cat filter.j2
{% set String = 'www|shencan|com' %}
{{ String | shencan('|') }}
{% set Time = '2015-06-22 18:48:01' %}
{{ Time | shencan1 }}
```

这里引入 shencan 和 shencan1 这两个自定义的 filter，下面我们把这个 Jinja2 放到 playbook 的 task 中运行：

```
[root@Master ~]# cat filter.yaml
---
- hosts: all
  tasks:
    - name: test jinja2 filter
      template: src=filter.j2 dest=/tmp/filter
```

```
[root@Master ~]# ansible-playbook filter.yaml -l 172.17.42.102
```

```

PLAY [all] *****

GATHERING FACTS *****
ok: [172.17.42.102]

TASK: [test jinja2 filter] *****
changed: [172.17.42.102]

PLAY RECAP *****
172.17.42.102      : ok=2      changed=1    unreachable=0    failed=0

[root@Master ~]# ansible 172.17.42.102 -a 'cat /tmp/filter'
172.17.42.102 | success | rc=0 >>

['www', 'shencan', 'com']
1434970081.0

```

因为 `str.split` 返回的格式是一个 `python list` 数据类型，所以 Jinja2 渲染后的结果直接通过 `list` 形式显示出来了。当然大家可以配合 Jinja2 的 `for` 语法进一步解析。这里就不做过多介绍了。关于 Jinja2 这个模板语言可以通过官网链接（<http://jinja.pocoo.org/docs/dev/api/#writing-filters>）进行深入学习。

## 6.6 本章小结

本章只是通过简单的例子去扩展 Ansible 的一些常用组件，通过前面的章节我们知道 Ansible 本身的扩展性非常强大，后面的章节也会介绍 Ansible 的一些常用 API。正是由于 Ansible 具有强大的扩展性，使得我们可以很容易地把 Ansible 和线上业务系统（包括监控系统）进行整合。

## 用 ansible-vault 保护敏感数据

通过前面章节的介绍，您应该已经深深体会到了 Ansible 是优秀的流程编排工具、入门门槛很低、简单易用，这都是为什么那么多人开始使用 Ansible 并喜欢上它的原因。但在运维工作中安全是非常重要工作之一，Ansible 又是如何保护敏感信息的呢？

Ansible 通过使用变量编写 `playbook`，在的变量使用中，我们看看如何分离数据和代码。通常敏感信息存储在数据中，如用户口令、数据基本凭证、API 密钥和其他与组织有关的特定信息。Ansible 的 `playbook` 作为代码通常保存在如 Git 这种版本控制系统中，这在协作的环境中很难保护敏感信息的安全。

从 Ansible 1.5 版本开始，Ansible 提供了 `vault` 数据安全解决方案。`vault` 采用经过验证的加密技术来保证敏感信息的安全存储和提取。使用 `vault` 的目标就是要对敏感信息进行加密，最大限度地保证数据在版本控制系统中的安全，并能够自由地存储和提取。

在本章中，我们将讲解下面三个方面内容：

- 了解 `ansible-vault` 如何保护数据。
- `ansible-vault` 保护数据的基本操作，包括加密、解密和重置密码操作。
- 介绍 3 个典型的实践场景，进一步掌握 `ansible-vault` 的实际使用。

## 7.1 了解 ansible-vault 如何保护数据

Ansible 提供了一个 `ansible-vault` 工具用于管理数据安全。`ansible-vault` 可以通过调用编辑器界面创建新的加密文件，也可以加密已经创建好了的文件。不管哪一种方式，都需要输入 `vault` 口令，这个口令采用 AES 加密算法加密数据。加密后的内容可以存储在版本控制系统，不会泄密。由于 AES 是基于公共对称密钥，因此在解密时候需要提供相同的口令。对于提供口令有两种方式，在执行 `ansible` 时候，通过 `--ask-vault-pass` 选项提示输入口令，或者通过 `--vault-password-file` 选项提供包含口令的完整路径的文件。

### 7.1.1 高级加密标准

高级加密标准 (Advanced Encryption Standard, AES)，在密码学中又称 Rijndael 加密法，是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的 DES，已经被多方分析且广为全世界所使用。经过长达五年的甄选流程，高级加密标准于 2001 年 11 月 26 日由美国国家标准与技术研究院 (NIST) 发布于 FIPS PUB 197，并在 2002 年 5 月 26 日成为有效的标准。2006 年开始，高级加密标准已成为对称密钥加密中最流行的算法之一。

AES 采用对称分组密码体制，密钥长度的最少支持为 128、192、256 位，分组长度 128 位，算法应易于各种硬件和软件实现。Ansible 使用的 AES 是 256 位最长密钥长度。

### 7.1.2 ansible-vault 能够加密什么

`ansible-vault` 能够加密任何结构化数据。由于 YAML 本身是一种结构化语言，几乎所有在 Ansible 编写的都满足这个标准。下面是一些能够用 `vault` 加密的内容：

- 用得最多的场景是加密变量，可以是：
  - `role` 中的变量文件，例如 `vars` 和 `defaults`。
  - 资源清单变量，例如 `host_vars`，`group_vars`。
  - 包含 `include_vars` 或 `vars_files` 的变量文件。
  - 通过 `-e` 选项参数传递给 Ansible-playbook 的变量文件，如 `-e @vars.yml` 或 `-e @vars.json`。
- 由于 `tasks` 和 `handlers` 也都是 JSON 数据，也可以用 `vault` 进行加密。然而，这在实际中非常少用。建议通过加密变量然后在 `tasks` 和 `handlers` 中引用。



下面是一些很好的加密对象：

- 凭证，例如数据库口令、应用凭证。
- API 密钥，例如远程访问密钥、私钥。
- Web 服务器的 SSL 密钥。
- 应用部署的 SSH 私钥。

当然还有些内容不适合做 vault 加密，例如：

- 由于 vault 加密的单位是基于文件，不能对部分文件或值进行加密。因此要么加密整个文件，要么不对文件加密。
- 文件和模板不能用 vault 加密，它们可能与 JSON、YML 文件不一样。

## 7.2 使用 ansible-vault

表 7-1 列出 ansible-vault 工具的子命令。

表 7-1 ansible-vault 工具的子命令

子命令	描述
create	使用编辑器创建加密文件。这需要在运行之前先配置编辑器的环境变量
edit	用编辑器编辑一个存在的加密文件，在内存中解密，退出编辑器后又保存成加密文件
encrypt	加密一个已有的结构化数据文件
decrypt	解密文件。使用这个命令要小心，不要把解密后的文件提交到版本控制系统中
rekey	改变用于加密、解密的口令

通过 ansible-vault 的 help 可以看到该命令的子命令，如下所示：

```
$ansible-vault --help
Usage: ansible-vault [create|decrypt|edit|encrypt|rekey|view] [--help]
[options] file_name
```

下面用实例介绍几个命令的具体用法。

### 7.2.1 创建加密数据文件

我们先创建一个加密文件，ansible-vault 用 create 子命令来创建新文件。用这个命令之前需要在环境变量中先设置编辑器，如下所示：

```
#setting up vim as editor
$export EDITOR=vim
#Generate a encrypted file
$ansible-vault create aws_creds.yml
Vault password:
Confirm Vault password:
```

在 vim 编辑器中输入访问的凭证，如图

7-1 所示。

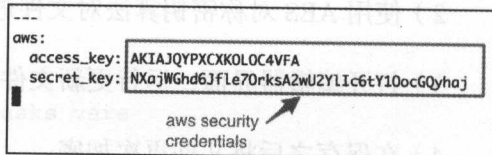


图 7-1 在 vim 编辑器中输入凭证

输入数据、保存、退出后，可以检查创建的文件类型以及文件内容，如下所示：

```
# Check file type and content
$ file aws_creds.yml
aws_creds.yml: ASCII text
$ cat aws_creds.yml
$ANSIBLE_VAULT;1.1;AES256
64616236666362376630366435623538336565393331333331663663636237636335313
234313134
3337303865323239623436646630336239653864356561640a363966393135316661636
562333932
61323932313230383433313735646438623032613635623966646232306433383335326
566343333
3136646536316261300a616438643463656263636237316136356163646161313365336
239653434
36626135313138343939363635353563373865306266363532386537623463623464376
134353863
37646638636231303461343564343232343837356662316262356537653066356465353
432396436
31336664313661306630653765356161616266653232316637653132356661343162396
331353863
34356632373963663230373866313961386435663463656561373461623830656261636
564313464
37383465353665623830623363353161363033613064343932663432653666633538
```

## 7.2.2 更新加密的数据文件

为了更新加密的 aws\_creds.yml，可以用 anisble-vault 的 edit 子命令进行修改，如下所示：

```
$ ansible-vault edit aws_creds.yml
Vault password:
```

edit 子命令将做以下操作：

- 1) 提示输入口令。
- 2) 使用 AES 对称密钥算法对文件进行解密。
- 3) 打开编辑器界面，运行更新文件的内容。
- 4) 在保存之后将文件再次加密。

还有另一种更新文件的方法，可以把文件先解密，如下所示：

```
$ ansible-vault decrypt aws_creds.yml
Vault password:
Decryption successful
```

更新完成后，再用 ansible-vault 加密文件方式进行加密。

### 7.2.3 变更加密数据密钥

良好的安全实践方式，通常是用 ansible-vault 来修改加密的密钥。这就需要 vault 对所有文件进行重新设置密钥。ansible-vault 提供了 rekey 子命令，使用如下：

```
$ ansible-vault rekey aws_creds.yml
Vault password:
New Vault password:
Confirm New Vault password:
Rekey successful
```

先询问当前口令，输入正确之后才运行你进行设置新的口令和再次确认新口令。注意，如果你使用版本控制系统管理这个文件，即使你文件的内容没有变化，也需要再次进行提交，重置密钥的操作将会更新最终加密的文件。

## 7.3 典型应用场景

下面介绍 ansible-vault 典型的应用场景。

### 7.3.1 实践场景 1: 保护 Ansible role 中的敏感数据

Ansible 可以加密 roles 中的敏感数据, 如 SSH 私有密钥可口令, 我们将在我们的服务器上用 Ansible 创建用户, 在此将用到 ansible-vault。

1) 创建 Users 角色 roles 目录结构如下:

```
$cd roles
$mkdir users
$cd users
$mkdir files handlers meta templates tasks vars
```

2) 定义该角色的依赖关系, 下面表示没有依赖:

```
---
dependencies: []
```

3) 变量。用 ansible-vault 创建 roles/users/variables 加密文件, 在 roles/users 目录下执行下面命令:

```
ansible-vault create vars/main.yml
```

输入 vault 的口令, 并再次输入校验下。之后就会打开选定的编辑器, 默认是 vim。

保存、退出 main.yml 编辑状态之后, 我们可以用 mkpasswd 命令使用 SHA-512 方法产生口令:

```
# Whois package contains "mkpasswd" command
sudo apt-get install -y whois
# Create a password for a user
mkpasswd --method=SHA-512
```

拷贝口令产生的 hash, 用 ansible-vault 的 edit 选项打开将对 vars/main.yml 文件进行编辑, 如下所示:

```
ansible-vault edit vars/main.yml
```

然后添加你的口令和其他敏感数据, 如下所示:

```
---
admin_password: <a generated password hash>
```

```

deploy_password: <another generated password hash>
shared_publickey: <your SSH public key to be placed in servers
authorized_keys directory>

```

保存、退出该文件，如果你编辑的时候没有带上 vault 选项，你将看到这个文件没有加密。

#### 4) 在任务中执行

现在已经创建了安全保护的变量，我们可以在 task 中使用它，创建 tasks/main.yml 任务文件：

```

---
- name: Create Admin User
  user: name=admin password="{{ admin_password }}" groups=sudo shell=/
    bin/bash
- name: Add Admin Authorized Key
  authorized_key: user=admin key="{{ shared_publickey }}"
    state=present
- name: Create Deploy User
  user: name=deploy password="{{ deploy_password }}" groups=www-data
    shell=/bin/bash
- name: Add Deploy Authorized Key
  authorized_key: user=deploy key="{{ shared_publickey }}"
    state=present

```

现在开始调整 nginx.yml 文件并命名为 servers.yml，然后执行：

```

---
- hosts: web
  sudo: yes
  user: root
  roles:
    - nginx
    - users

```

测试并运行，如下所示：

```

ansible-playbook --syntax-check --ask-vault-pass servers.yml
ansible-playbook --ask-vault-pass --private-key=~/.ssh/id_ansible
  servers.yml

```



现在可以登录远程的节点，并检查我们的服务器：

```
# Check for user "admin" and "deploy"
cat /etc/passwd
```

从本地的计算机，用新用户尝试登录，如下：

```
# For me, logging in using the Ansible SSH key looked like this:
ssh -i ~/.ssh/id_ansible admin@104.131.43.90
```

### 7.3.2 实践场景 2：使用加密做用户认证

授权在 Juniper 网络上执行用 Ansible 模块保存的口令，你也可以创建 ansible-vault 来保存口令。创建和使用 ansible-vault 加密的文件都需要设置口令。

1) 创建一个 vault 加密过的数据文件，并通过指定加密、解密的口令、编辑修改数据文件，如下所示：

```
[root@ansible-cm]# ansible-vault create vault-vars.yml
Vault password:
Confirm Vault password:
```

2) 加密的 ansible-vault 编辑了所有需要的变量，并保存：

```
[root@ansible-cm]# ansible-vault edit vault-vars.yml
Vault password:
PORT_NETCONF: 830
PORT_TELNET: 23
ROOT_USER: root
ROOT_PASSWORD: password
```

3) 查看有 ansible-vault 生成的加密文件如下：

```
[root@ansible-cm]# cat vault-vars.yml
$ANSIBLE_VAULT;1.1;AES256
31415961343966623035373532313264333633663764353763393066643131306565636
463326634
3730326165666565356665343137313161234569336336640a653939633331663935376
362376666
65653737653262363235353261626135312345663665396262376339623737366238653
436306663
```

```
6430376633306339360a343065363331313532633036343866376330623634653538353
132314159
3835
```

4) 在 playbook 中, 包含了 vault 加密的变量文件, 需要的时候可以引用合适的变量:

```
---
- name: Get Device Facts
  hosts: dc1
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
  vars_files:
    - vault-vars.yml
  tasks:
    - name: Checking NETCONF connectivity
      wait_for: host={{ inventory_hostname }} port={{ PORT_NETCONF }}
      timeout=5
    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
        user={{ ROOT_USER }}
        passwd={{ ROOT_PASSWORD }}
      register: junos
    - name: version
      debug: msg="{{ junos.facts.version }}"
```

5) 执行 playbook 时带上 --ask-vault-pass 选项, 将会提示输入口令:

```
[root@ansible-cm]# ansible-playbook playbook-name.yml --ask-vault-pass
Vault password:
```

```
PLAY [Get Device Facts] *****
```

```
...
```

### 7.3.3 实践场景 3: 保护 Nginx 中的 SSL 密钥

前面我们说过, 模板可能与 YAML 或 JSON 文件不同, 不一定是结构化文件, 不

能直接进行加密。但是有一种方法可以把加密的数据嵌入模板中，毕竟模板是动态加载内容的，这些动态内容是来自于变量，变量是可以加密的。下面来讨论如何实现在 Nginx Web 服务中添加 SSL 支持。

我们已经搭建好了 Nginx Web 服务，现在我们来添加 SSL 的支持，需要完成下面过程。

### 1) 我们按照下面添加变量：

```
#file: roles/nginx/defaults/main.yml
nginx_ssl: true
nginx_port_ssl: 443
nginx_ssl_path: /etc/nginx/ssl
nginx_ssl_cert_file: nginx.crt
nginx_ssl_key_file: nginx.key
```

### 2) 创建自签名的 SSL 证书：

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout nginx.key -out
  nginx.crt
```

这条命令将会生产 nginx.key 和 nginx.crt 两个文件，将要被拷贝到 Web 服务器上。

### 3) 将这些文件的内容添加到变量中，并创建 group\_vars/www 文件：

```
# file: group_vars/www
---
nginx_ssl_cert_content: |
    -----BEGIN CERTIFICATE-----
    -----END CERTIFICATE-----
nginx_ssl_key_content: |
    -----BEGIN PRIVATE KEY-----
    -----END PRIVATE KEY-----
```

这实例中只是列出了占位，真实环境需要替换成实际密钥和证书，同时这证书和凭证不能直接保存在版本控制系统中。

### 4) 用 ansible-vault 加密这变量文件：

```
$ ansible-vault encrypt group_vars/www
```

```
Encryption successful
```

我们在配置文件中设置了 vault 口令的位置，因此 ansible-vault 不再询问口令。

### 5) 创建添加密钥之后的模板如下：

```
# filename: roles/nginx/templates/nginx.crt.j2
{{ nginx_ssl_cert_content }}
# filename: roles/nginx/templates/nginx.key.j2
{{ nginx_ssl_key_content }}
```

### 6) 也添加一个支持 SSL 的虚拟主机的配置文件：

```
# filename: roles/nginx/templates/nginx.key.j2
server {
    listen {{ nginx_port_ssl }};
    server_name {{ ansible_hostname }};
    ssl on;
    ssl_certificate {{ nginx_ssl_path }}/{{ nginx_ssl_cert_file }};
    ssl_certificate_key {{ nginx_ssl_path }}/{{ nginx_ssl_key_file }};
    location / {
        root {{ nginx_root }};
        index {{ nginx_index }};
    }
}
```

7) 还需要创建个配置 SSL 站点的 task 文件，需要创建必要的目录、文件和配置，如下所示：

```
---
# filename: roles/nginx/tasks/configure_ssl.yml
- name: create ssl directory
  file: path="{{ nginx_ssl_path }}" state=directory owner=root
  group=root
- name: add ssl key
  template: src=nginx.key.j2 dest="{{ nginx_ssl_path }}/nginx.
    key" mode=0644
- name: add ssl cert
  template: src=nginx.crt.j2 dest="{{ nginx_ssl_path }}/nginx.
    crt" mode=0644
- name: create ssl site configurations
  template: src=default_ssl.conf.j2 dest="{{ nginx_ssl_path }}/
```

```

    default_ssl.conf" mode=0644
  notify:
    - restart nginx service

```

8) 最后, 根据 `nginx_ssl` 参数, 调用这个任务选项:

```

# filename: roles/nginx/tasks/main.yml
- include: configure_ssl.yml
  when: nginx_ssl

```

9) 按照下面执行 `playbook`:

```
$ ansible-playbook -i customhosts site.yml
```

这个自签名支持 SSL 的站点需要运行在 443 端口, 现在应该可以用 HTTPS 安全协议打开 Web 服务器地址。

当然, 这将会显示告警, 我们的证书是自签名, 不是签名认证机构签发的。

## 7.4 本章小结

本章讲解了如何使用 `ansible-vault` 来保证 `playbook` 中数据的安全。我们从需要加密数据的需求开始, 介绍 `ansible-vault` 是如何工作、如何加密的。然后详细介绍 `ansible-vault` 工具的基本操作, 如创建加密文件、解密文件、重置密钥等。接着介绍了如何加密已有的文件, 如运行 `ansible-vault` 对 `vars` 文件加密来保证数据库凭证的安全。最后介绍了在支持 SSL 的 Nginx 环境中, 如何用 `vault` 实现安全存储私有密钥和凭证, 以及通过模板进行复制。当然 `ansible-vault` 提供的是一种支持 Ansible 数据的安全模型, 在使用 `ansible-vault` 时候, 还需要对这里没有介绍的系统进行安全加固。



## Ansible 与云计算

在我们日常工作中一般都是使用 Ansible 当作配置管理工具，而实际上 Ansible 不仅是一款优秀的配置管理工具，它还是架构编排的利器。通常我们都是对现有的设备去进行配置管理，其实 Ansible 还支持各大云平台的架构编排。可能国内的运维人员很少接触国外的公有云平台，但是大家肯定对 OpenStack、AWS 和 Docker 这些云计算技术很熟悉。我们在查看 Ansible 的自带模块时会发现 Ansible 对云计算这块是支持最多的一个配置管理工具。本章将分别介绍 Ansible 结合 AWS 和 OpenStack 自动创建配置实例，以及对目前市场上最火的 Docker 进行日常管理，最后再介绍 Ansible 和 jenkins 实现一个自动化持续集成的项目。

### 8.1 了解云平台管理流程

通常我们在使用 Ansible 的时候首先需要知道一台机器的 IP 地址以及 SSH 认证方式信息等，然后 Ansible 通过 SSH 方式去管理它。Ansible 的云计算中的应用场景都是在本地使用 Ansible 已经封装好的云平台模块（API）去连接到云平台上新建一个或者多个实例，其自动根据云平台生成后的实例信息做下一步的任务，比如新建机器设备信息录入 CMDB 系统，对新建的机器进行自动化部署，对新建的机器添加监控，等等。这样 Ansible 可以很快地新建出一套业务架构或者对现有业务架构进行快速扩展。

## 8.2 Ansible AWS 和 OpenStack

AWS 是亚马逊公司的云计算 IaaS 和 PaaS 平台，提供了一整套基础设施和应用程序服务。你能够在云中运行一切应用程序，而且 AWS 提供各种服务。OpenStack 也是目前最流行的一个开源的云计算管理平台项目，使用它可以很容易搭建一套企业级别的云基础架构服务平台。由于篇幅有限，本书就不针对 AWS 和 OpenStack 技术进行深入浅出讲解了。下面笔者就通过曾经使用 Ansible 管理 AWS 和 OpenStack 的一个项目进行讲解 Ansible 是如何跟 AWS 和 OpenStack 紧密工作的。

### 1. Ansible 与 AWS

熟悉了流程之后我们就通过一些示例来讲解如何使用 Ansible 来管理 AWS。代码如下：

```
---
- name: new aws instances
  hosts: 127.0.0.1
  connection: local
  gather_facts: False
  vars:
    ID: "AKIAI5F43DUPLEVF44WS"
    KEY: "3f6mzQAhiL0gnlgPaaenttyD3MfVv3Fwm9xsde"
  tasks:
    - name: allocated EIP
      local_action: allocated region={{ re }}
      register: EIP
      environment:
        AWS_ACCESS_KEY_ID: "{{ ID }}"
        AWS_SECRET_ACCESS_KEY: "{{ KEY }}"
    - name: new instances when env=production
      local_action:
        module: ec2
        region: "{{ re }}"
        keypair: "Yottaa-{{ key }}-v1307"
        group_id: "{{ sid }}"
        instance_type: "{{ type }}"
        image: "{{ ami }}"
        vpc_subnet_id: xxxxxxxx
        assign_public_ip: yes
```

```
aws_access_key: "{{ ID }}"
aws_secret_key: "{{ KEY }}"
```

```
wait: yes
```

```
volumes:
```

- device\_name: /dev/sdb
  - ephemeral: ephemeral0
- device\_name: /dev/sdc
  - ephemeral: ephemeral1
- device\_name: /dev/sdd
  - ephemeral: ephemeral2
- device\_name: /dev/sde
  - ephemeral: ephemeral3

```
register: ec2production
```

- name: add tags to instances when env=production

```
local_action: ec2_tag resource={{ item.id }} region={{ re
}} aws_access_key={{ ID }} aws_secret_key={{ KEY }}
state=present
```

```
with_items: ec2production.instances
```

```
args:
```

```
tags:
```

```
Name: "{{ ro | upper }}"
```

- name: associate the EIP

```
local_action: associate region={{ re }} allocateid={{ EIP.
allocateid }} instanceid={{ item.id }}
```

```
with_items: ec2production.instances
```

```
environment:
```

```
AWS_ACCESS_KEY_ID: "{{ ID }}"
```

```
AWS_SECRET_ACCESS_KEY: "{{ KEY }}"
```

- name: add instances to hosts group\_id when env=production

```
local_action: add_host hostname={{ EIP.eip }} ansible_
ssh_private_key_file=./Yottaa-Prod-v1307.pem ansible_
ssh_user=yadmin groups=aws
```

- name: create body files

```
local_action: template src=./body-{{ env }}.j2 dest=../{{
item.id }}.json
```

```
with_items: ec2production.instances
```

```

- name: insert data to CMDB
  uri: url=http://ops.yottaa.com:8000/api/instance/instances/
      method=POST user=api password=qwl2ewq
      body='{{ lookup('file','.'/' + item.id + '.json') }}' force_
      basic_auth=yes
      status_code=201 HEADER_Content-Type="application/json"
  with_items: ec2production.instances

- name: wait for SSH to running when env=stage
  local_action: wait_for port=22 host={{ EIP.eip }} delay=10

- name: configure the machine
  hosts: aws
  user: yadmin
  sudo: yes
  tasks:
    - include: init.yml

```

这是一个最简单的使用 Ansible 管理 AWS 的 playbook。因为这个 playbook 只针对本机运行，所以 hosts 是本机而且 connection 方式是 local，tasks 都是使用 local\_action 模式。还有使用 AWS 的 API（模块就是封装 Python 下的 boto 库，需要提前安装）需要 AWS 的 ID 和 ACCESS\_KEY，这个可以通过 AWS console 生成即可。我们主要看第 2 个 task，这里使用了 Ansible 的 ec2 模块去新建机器，指定了该机器的一些属性，比如 region、vpc\_id、安全组 id 等信息。此 task 运行后，AWS 会把新建这台机器的所有信息返回给 ec2production。下面我们就可以针对这台机器做如下一些相关操作，例如：

- 第 3 个 tasks 给这台新机器指定一个 tag 标签。
- 第 4 个 task 给这台机器分配一个 EIP 地址。
- 第 5 个 task 是把这台新建的主机使用 add\_host 模块添加到 Inventory 里面并且指定一些 SSH 认证信息（临时添加）。
- 第 6 个和第 7 个 task 是把新生成的机器的信息进行整理然后通过 CMDB 系统的 API 自动加入到 CMDB 系统。
- 第 8 个 task 是等待这台新的机器 SSHD 服务运行正常后，进行下一步处理。
- 最后一个 task 是针对上面的机器进行配置管理了。引用的 playbook 已经封装好了很多配置流程，所以只要这个 playbook 运行完成后，一台机器所有的部署工作都自动化完成了，下一步的工作就是随时把它加入到业务系统中。

## 2. Ansible 与 OpenStack

通过上面同样的方法我们也可以使用 Ansible 在 OpenStack 平台上新建并且配置机器，下面我们直接通过示例讲解。当然每个模块的依赖包需要提前安装好，比如 python-novaclient python-keystoneclient 等。

```

---
- name: new openstack instances
  hosts: 127.0.0.1
  gather_facts: False
  vars:
    auth_url: http://10.8.6.2:35357/v2.0/
    login_username: admin
    login_tenant_name: admin
    login_password: yottaaops!@#
    image_id: clc28aa4-a892-40a0-b7aa-f692fd2785b1
    keypair_name: Yottaa-Stage-v1307
    private_net: 696bfa85-5e8e-4dea-b41c-e066bfa925ac
  connection: local
  tasks:
    - nova_compute:
      auth_url: "{{ auth_url }}"
      login_username: "{{ login_username }}"
      login_password: "{{ login_password }}"
      login_tenant_name: "{{ login_tenant_name }}"
      security_groups: default
      state: present
      name: "{{ ro }}-{{ 1000000 | random(1, 1) }}.yottaa.com"
      image_id: "{{ image_id }}"
      key_name: "{{ keypair_name }}"
      flavor_id: "{{ flavor_id }}"
      nics:
        - net-id: "{{ private_net }}"
      register: openstack

- name: pull floating ip address
  local_action: floating
    username={{ login_username }} password={{ login_password }}
    tenant={{ login_tenant_name }} authurl={{ auth_url }}

```



```

register: floating_ip

- name: bond floating ip to instance
  local_action: bondip_openstack
    username={{ login_username }} password={{ login_password }}
    tenant={{ login_tenant_name }} authurl={{ auth_url }}
    instance_id={{ openstack.id }} floating_ip={{ floating_ip.res }}

- name: add instances to hosts
  local_action: add_host hostname={{ floating_ip.res }} ansible_ssh_private_key_file=/Users/shencan/key/Yottaa-Stage-v1307.pem ansible_ssh_user=root groups=openstack

- name: wait for SSH to running
  local_action: wait_for port=22 host={{ floating_ip.res }} delay=10

- name: create body files
  local_action: template src=./body-openstack.j2 dest=/tmp/{{ openstack.id }}.json

- name: insert data to CMDB
  uri: url=http://ops.yottaa.com:8000/api/instance/instances/
    method=POST user=api password=qw12ewq
    body='{{ lookup('file','/tmp/' + item + '.json') }}' force_basic_auth=yes
    status_code=201 HEADER_Content-Type="application/json"
  with_items: openstack.id

- name: sendmail
  local_action: mail
    host='127.0.0.1'
    port=25
    subject="new instances information in {{ openstack.info.name }}"
    body="instances id is {{ openstack.id }} public_

```

```

2. Ansible 与 OpenStack    ip is {{ floating_ip.res }} "
                           to="xiangjun.zhang@yottaa.com,can.shen@yottaa.com"
                           charset=utf8
                           attach="/tmp/{{ openstack.id }}.json"

```

```

- name: configure the machine
  hosts: openstack
  user: root
  tasks:
    - include: init.yml

```

可以看到整个 playbook 流程跟 AWS 流程是一样的，需要先通过 nova\_compute 模块去创建实例，然后根据 OpenStack 返回的信息进行操作。所以这个 playbook 就不再进行详细讲解了。

## 8.3 Ansible 与 Docker

Docker 是目前最热门的一个虚拟化管理工具，相当于 Xen 或者 KVM 虚拟化技术的 Docker 有着它独一无二的优势。关于 Docker 的安装，本书不再进行介绍了。如果 Docker 安装的依赖都满足了，直接使用 Ansible IP -m yum -a 'name=docker-io state=latest' 安装即可。目前 Ansible 对 Docker 只有两个模块，一个是对 Docker 镜像的日常管理模块 docker\_image，一个是对 Docker 容器的日常管理模块 Docker。关于这两个模块的用法可以使用 ansible-doc docker 或者 ansible-doc docke\_iamge 进行查看。当然也要了解这两个模块的依赖，安装 docker-py python 库即可。如果想远程使用 Ansible 管理 Docker，一定得让 dokcer 服务监听一个 TCP 端口。下面我们就分别通过示例来讲解如何使用 Ansible 管理 Docker。

### 1. Docker 镜像管理

我们先来看一下 playbook：

```

[root@vm10-160-112-18 ~]# cat docker_imeage.yaml
---
- hosts: 127.0.0.1
  tasks:
    - name: build image
      docker_image: path="/root/Dockerfiles" name="centos6.6/ssh"
                   tag="v5" state=present

```

这个 playbook 在本机上建立一个 Docker 镜像，当然也支持远程管理，通过 `docker_url` 参数指定远端 Docker 服务器即可。下面我们运行这个 playbook，然后查看 Docker 镜像：

```
[root@vm10-160-112-18 ~]# ansible-playbook docker_image.yml

PLAY [127.0.0.1] *****

GATHERING FACTS *****
ok: [127.0.0.1]

TASK: [build image] *****
ok: [127.0.0.1]

PLAY RECAP *****
127.0.0.1          : ok=2    changed=0    unreachable=0    failed=0
```

```
[root@vm10-160-112-18 ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
centos6.6/ssh	v5	6479cea71ed5	5 days ago
		346.3 MB	

Dockerfiles 目录下的文件如下所示：

```
[root@vm10-160-112-18 Dockerfiles]# tree
```

```
.
├── Dockerfile
├── epel-release-6-8.noarch.rpm
├── ssh.repo
└── supervisord.conf
```

```
0 directories, 4 files
```

```
[root@vm10-160-112-18 Dockerfiles]# cat Dockerfile
```

```
FROM centos:6.6
```

```
MAINTAINER shencan http://www.shencan.net
```

```
# ----- Installing Salt -----
```

```
ADD ./epel-release-6-8.noarch.rpm /root/epel-release-6-8.noarch.rpm
```

```

ADD ./ssh.repo /etc/yum.repos.d/ssh.repo
RUN rpm -vih /root/epel-release-6-8.noarch.rpm
RUN yum install salt-minion openssh-server supervisor openssh-clients
    openssh telnet -y
RUN yum install --assumeyes which
RUN sed -i 's/#master: salt/master: 192.168.2.1/g' /etc/salt/minion
RUN echo 123456 |passwd --stdin root
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
RUN ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key
RUN sed -i 's/UsePAM yes/UsePAM no/g' /etc/ssh/sshd_config
ADD supervisord.conf /etc/supervisord.conf
EXPOSE 22
CMD ["/usr/bin/supervisord"]

```

这里只是简单地使用 Ansible 去建立了一个镜像，关于 Docker 镜像的其他管理方式 Ansible 也是支持的，比如删除一个镜像。

## 2. Docker 容器管理

介绍了 Docker 镜像的管理后，我们再来介绍使用 Ansible 来管理 Docker 容器，比如新建或删除容器等操作。我们通过上一节建立的镜像生成三个容器，如下所示：

```

---
- hosts: 127.0.0.1
  tasks:
    - name: docker container
      docker: name=ansible{{ item }} image=centos6.6/ssh:v5 state=
        started docker_api_version=1.7.1
      with_items:
        - 1
        - 2
        - 3

```

这就是一个简单的运行 docker 容器，当然 docker 模块也支持很多其他参数，比如端口映射、目录挂载等。具体参数通过 `ansible-doc docker` 查看即可。我们运行这个 playbook，然后查看 docker 容器，如下所示：

```
[root@vm10-160-112-18 ~]# ansible-playbook docker.yaml
```

```
PLAY [127.0.0.1] *****
```

```
GATHERING FACTS *****
```

```
ok: [127.0.0.1]
```

```
TASK: [docker container] *****
changed: [127.0.0.1] => (item=1)
changed: [127.0.0.1] => (item=2)
changed: [127.0.0.1] => (item=3)
```

```
PLAY RECAP *****
127.0.0.1          : ok=2    changed=1    unreachable=0    failed=0
```

```
[root@vm10-160-112-18 ~]# docker ps |grep ansible
2b266ad450eb        centos6.6/ssh:v5    "/usr/bin/supervisor    9 seconds
ago                Up 9 seconds        22/tcp                ansible3
ef9adca84a8d        centos6.6/ssh:v5    "/usr/bin/supervisor    10
seconds ago        Up 10 seconds        22/tcp                ansible2
21aca76726e5        centos6.6/ssh:v5    "/usr/bin/supervisor    11
seconds ago        Up 11 seconds        22/tcp                ansible1
```

## 8.4 Ansible Jenkins

Jenkins 是一款基于 Java 开发的持续集成工具，Jenkins 提供一个友好的平台可以让我们去完成那些重复的工作，目前 Jenkins 已经支持各种各样的插件，包括 SaltStack、Ansible、Puppet 这类自动化配置管理工具。我们可以把经常重复的工作集成到 Jenkins 上，从而提高工作效率。Jenkins 是一个强大的平台，它可以配合强大的插件库实现很多功能，本书只是简单讲解 Jenkins 与 Ansible 的集成，关于 Jenkins 更多强大的功能可以参考 Jenkins 官网 (<http://jenkins-ci.org/>)。

### 1. 安装 配置 Jenkins

Jenkins 安装方式很多，本书只介绍在 CentOS 下采用 yum 安装方式，其他安装方式可以参考 Jenkins 官网。首先我们需要添加 Jenkins 软件源，然后使用 yum install jenkins -y 即可。需要注意的是 Jenkins 依赖 Java 环境，所以安装 Jenkins 之前确保 Java 环境已经安装，如果没有 Java 环境也可以使用 yum install java-1.7.0-openjdk -y 安装。下面是导入 Jenkins 源：

```
wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/
jenkins.repo
rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
```



默认 Jenkins 服务是以 Jenkins 用户运行的，包括在 Jenkins 平台上的项目都是在 Jenkins 用户下运行的，这里为了下面调用 Ansible 插件，所以改成 root 用户了。最后启动服务即可：

```
[root@vm10-160-112-18 ~]# cat /etc/sysconfig/jenkins |grep USER
JENKINS_USER="root"
[root@vm10-160-112-18 ~]# /etc/init.d/jenkins start
Starting Jenkins [ 确定 ]
```

然后通过 <http://IP:8080> 访问 Jenkins UI。第一次登录时不需要任何认证信息，登录进去后通过“系统管理->Configure Global Security”设置用户名与密码。Jenkins 也支持 LDAP 认证。

## 2. 安装 Jenkins Ansible 插件

到这里 Jenkins 平台的基础环境已经搭建好了，下面我们就介绍如何与 Ansible 结合。目前 Jenkins 已经有插件支持 Ansible 了，插件名称是 Ansible Plugin，关于插件相关介绍可以通过以下地址进行了解：

<https://wiki.jenkins-ci.org/display/JENKINS/Ansible+Plugin>

下面我们就通过 Jenkins 平台安装这个插件，点击“系统管理->管理插件->可选插件”，然后搜索 Ansible，勾选 Ansible plugin，点击直接安装即可。为了对 Ansible 执行结果颜色格式化，这里我们顺便也安装一个 AnsiColor 插件。安装方法与上面一样，如图 8-1 所示。安装好插件后，需要对插件进行相应的配置。点击“系统管理->系统设置->Ansible”即可。这里我们需要指定 Ansible 和 ansible-playbook 执行文件的目录。最后点击“保存”。

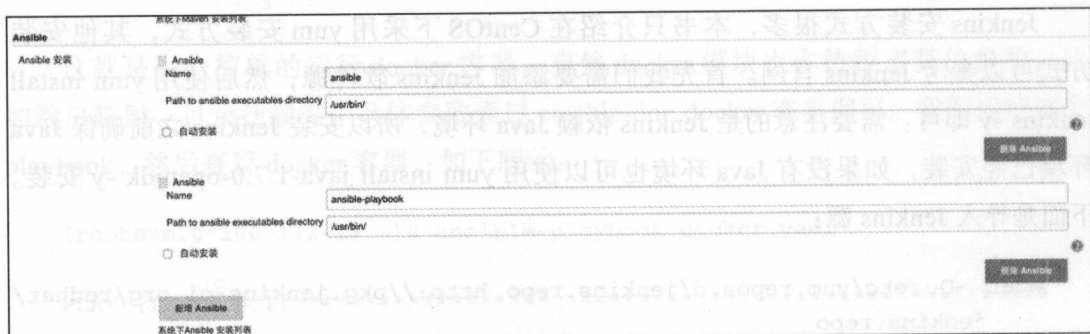


图 8-1 ansible-playbook 执行文件定义

到这里 Ansible 插件的安装与配置就完成了。下面我们就可以建 Jenkins 项目了。

### 3. 新建 Ansible 项目

现在我们建两个 Ansible 项目：一个是执行 Ansible Ad-Hoc 命令，一个是执行 ansible-playbook。新建项目时直接点击“新建”按钮即可。这里我们选择构建一个自由风格的软件项目，第一个项目我们起名叫 Ansible Ad-Hoc Command。然后进入项目配置页面。这里我们只需关心构建。点击增加构建步骤，如图 8-2 所示，我们选择 Invoke Ansible Ad-Hoc Command。然后参数如下，最后点击“保存”。

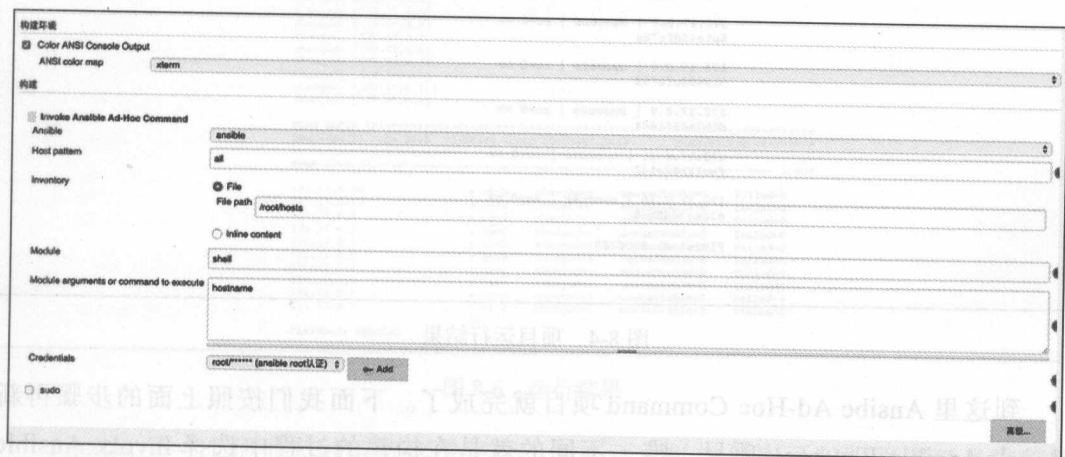


图 8-2 构建项目配置

Ansible Ad-Hoc Command 项目就建好了。我们点击构建就可以运行项目了。项目运行完成后，点击 console output 查看结果，如图 8-3 和图 8-4 所示。

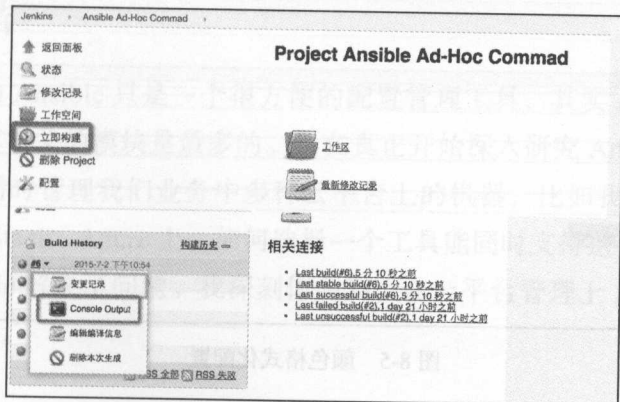


图 8-3 构建运行项目

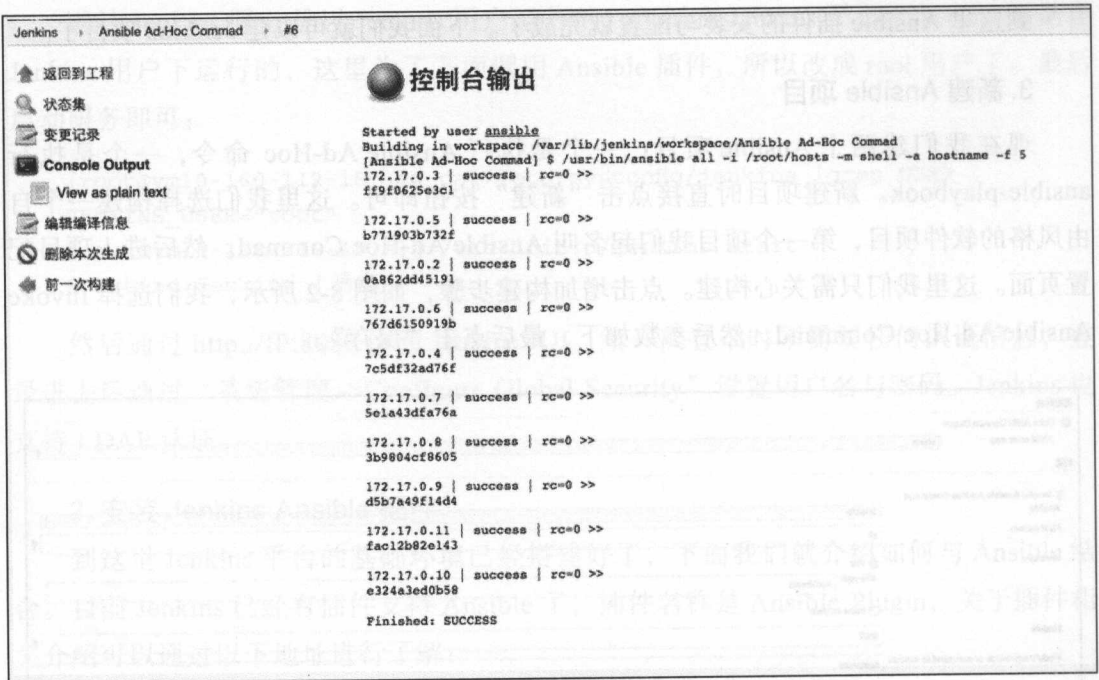


图 8-4 项目运行结果

到这里 Ansible Ad-Hoc Command 项目就完成了。下面我们按照上面的步骤再新建一个 Ansible Playbook 项目，唯一不同的就是在构建的过程中选择 Invoke Ansible Playbook。然后制定下 playbook 文件即可，颜色格式化配置如图 8-5 所示。同样最后点击“保存”。

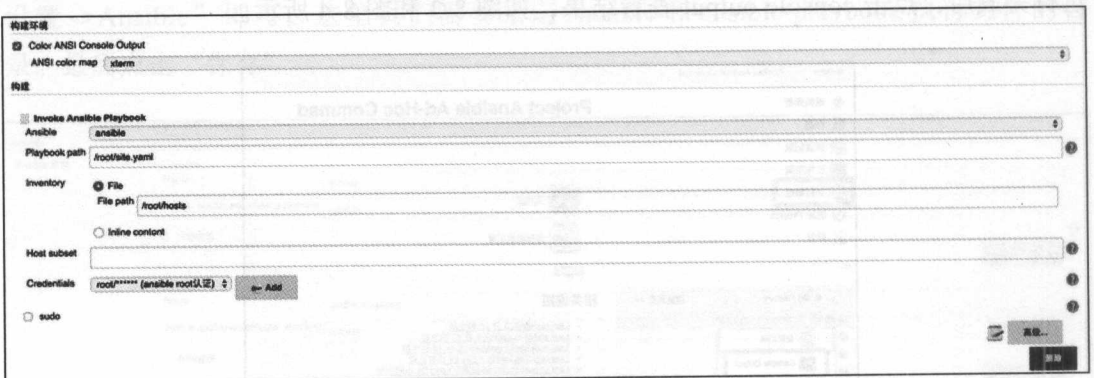


图 8-5 颜色格式化配置

接着我们也点击立即构建运行 playbook，最后查看结果，如图 8-6 所示。

```

Jenkins > Ansible playbook > #21

控制台输出

Started by user ansible
Building in workspace /var/lib/jenkins/workspace/Ansible playbook
[Ansible playbook] $ sshpass -U ***** /usr/bin/ansible-playbook /root/site.yaml -i /root/hosts -f 5 -u root -k

PLAY [all] *****
GATHERING FACTS *****
Thursday 02 July 2015 22:54:40 +0800 (0:00:00.015) 0:00:00.015 *****
ok: [172.17.0.3]
ok: [172.17.0.5]
ok: [172.17.0.4]
ok: [172.17.0.2]
ok: [172.17.0.6]
ok: [172.17.0.7]
ok: [172.17.0.8]
ok: [172.17.0.10]
ok: [172.17.0.9]
ok: [172.17.0.11]

TASK: [test] *****
Thursday 02 July 2015 22:54:42 +0800 (0:00:01.528) 0:00:01.544 *****
changed: [172.17.0.4]
changed: [172.17.0.5]
changed: [172.17.0.6]
changed: [172.17.0.2]
changed: [172.17.0.7]
changed: [172.17.0.8]
changed: [172.17.0.10]
changed: [172.17.0.9]
changed: [172.17.0.11]
changed: [172.17.0.3]

PLAY RECAP *****
Thursday 02 July 2015 22:54:43 +0800 (0:00:01.394) 0:00:02.939 *****
test ----- 1.40s
172.17.0.10      : ok=2    changed=1    unreachable=0    failed=0
172.17.0.11      : ok=2    changed=1    unreachable=0    failed=0
172.17.0.2       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.3       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.4       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.5       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.6       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.7       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.8       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.9       : ok=2    changed=1    unreachable=0    failed=0
172.17.0.11      : ok=2    changed=1    unreachable=0    failed=0

Finished: SUCCESS

```

图 8-6 运行结果

Jenkins 是一个很强大的持续集成平台，它有强大的插件库可以跟很多工具进行集成。本书只对 Jenkins 进行最简单的介绍，强烈推荐读者去认真去了解 Jenkins 这个强大的持续集成工具。

## 8.5 本章小结

可能有人认为 Ansible 只是一个很方便的配置管理工具，其实 Ansible 所有自带的模块中关于云计算相关的模块是最多的。我在真正开始深入研究 Ansible 之前，最早有个需求就是能够同时管理我们业务中多种云平台上的机器，比如我就遇到有业务同时跑在 AWS、OpenStack、Azure 上，如何选择一个工具能同时支持这些云平台很费周折，后来我选择 Ansible 解决了问题。我深刻体会到，在云平台管理上 Ansible 具有很多工具难以媲美的功能。

## 部署 Zabbix 组件

Zabbix 是目前最流行的一款企业级别的监控软件，它的易用性以及扩展性非常强大，加上自带丰富的 API 功能，使得它几乎成为每个互联网公司的标配运维组件。Zabbix 的默认架构是 C/S 架构，当然 Zabbix 也支持 Proxy 代理架构。关于 Zabbix 的其他相关知识可以参考我的好友吴兆松同学编写的《Zabbix 企业级分布式监控系统》这本书籍，我个人认为这是市面上最好的 Zabbix 书籍。本章将介绍用 Ansible 配置部署 Zabbix。

### 9.1 了解部署流程

在使用 Ansible 配置部署 Zabbix 之前，我们一定得先清楚想要实现一个什么样的环境，比如我们使用 Ansible 安装 zabbix-server、zabbix-proxy 和 zabbix-agent，我们希望 Ansible 完成运行后，整个 Zabbix 环境处于一个什么状态以及后续去怎么维护和扩展，等等。因为 Zabbix 有数据库、server 和 proxy 以及 agent 的概念，所以这里我们会涉及配置文件中每个业务角色的定义和指向问题。Zabbix 最简单的流程就是 agent 端配置中指向 proxy 服务器，proxy 服务器会处理该节点所有 agent 的监控数据以及存储，最后会把所有节点的数据发送给 server 端，然后 server 对数据进行存储分析以及相关的展示，等等。



在了解整个业务逻辑之后，我们就可以规划如何使用 Ansible 来实现这个状态或者功能。

## 9.2 编写业务 roles

为了更好地管理和复用 Ansible 的配置管理文件，笔者使用 roles 的形式编写日常配置管理文件。通过上一节分析 Zabbix 数据流程之后，我们就可以知道需要给哪些业务角色编写 playbook 了。

这里是针对 Zabbix 业务角色来编写 role，比如笔者准备了 4 个 roles 分别是 base、zabbix-server、zabbix-proxy、zabbix-agent，在实际部署过程中会根据需求部署来引用对应的 role。base 角色的工作主要是针对所有主机进行一些基础环境初始化。

下面我们为 Zabbix 环境进行主机信息相关的规划（见表 9-1），操作系统采用最新的 CentOS 6.7，zabbix-server、zabbix-proxy、zabbix-agent 都是采用目前最新的 2.4.6 版本。

表 9-1 主机信息

主机名	ip 地址	角色
server.shencan.net	192.168.1.100	zabbix-server/mysql-server/zabbix-agent/zabbix-ui
proxy.shencan.net	192.168.1.115	zabbix-proxy/mysql-server/zabbix-agent
agent.shencan.net	192.168.1.111	zabbix-agent

主机信息规范好之后，就可以在 Ansible 的 hosts 文件定义主机以及相关的变量了：

```
[zabbix-server]
192.168.1.100    hostname=server.shencan.net
[zabbix-proxy]
192.168.1.115    hostname=proxy.shencan.net
[zabbix-agent]
192.168.1.111    hostname=agent.shencan.net
```

为了以后扩展以及批量部署 zabbix-agent，这里在 group\_vars/all 变量定义文件中定义了 zabbix-server 和 zabbix-proxy 相关的主机信息：

```
---
zabbix_server: 192.168.1.100
```

```
zabbix_proxy: 192.168.1.115
ansible_ssh_user: root
```

hosts 文件和主机变量信息定义好之后就可以编写 role 了。下面将分别介绍每个 role 相关的 playbook 以及模板文件。我们首先从 base 角色开始。来看看 main.yaml 文件：

```
---
- name: set hostname
  hostname: name={{ hostname }}

- name: Change network files
  shell: sed -i "s/HOSTNAME=.*HOSTNAME={{ hostname }}/g" /etc/
        sysconfig/network

- name: Stop Iptables
  service: name=iptables state=stopped enabled=no

- name: disable selinux
  shell: /usr/sbin/setenforce 0 && sed -i 's/SELINUX=enforcing/
        SELINUX=disabled/g' /etc/sysconfig/selinux

- name: Install libselinux-python
  raw: yum install libselinux-python -y

- name: copy epel yum source
  copy: src={{ item.src }} dest={{ item.dest }} owner=root group
        =root mode=644
  with_items:
    - {src: epel.repo, dest: /etc/yum.repos.d/epel.repo }
    - {src: RPM-GPG-KEY-EPEL-6, dest: /etc/pki/rpm-gpg/RPM-GPG
      -KEY-EPEL-6 }
    - {src: RPM-GPG-KEY-ZABBIX, dest: /etc/pki/rpm-gpg/RPM-GPG
      -KEY-ZABBIX }
    - {src: zabbix.repo, dest: /etc/yum.repos.d/zabbix.repo }

- name: copy /etc/hosts files
  template: src=hosts.j2 dest=/etc/hosts owner=root group=root
            mode=644
```

第 1 个 task 是引用 hostname 模块进行主机名的修改。

第 2 个 task 是引用 shell 模块进行 /etc/sysconfig/network 文件中主机名的修改。

第3个 task 是引用 service 模块关闭 iptables 服务。

第4个 task 是引用 shell 模块对 selinux 的关闭以及配置。

第5个 task 是引用 raw 模块进行安装 libselinux-python 包（如果主机开启了 selinux 需要安装此包）。

第6个 task 是引用 copy 模块进行 yum 源配置的同步，这里采用了 with\_items 进行多个文件循环同步。

第7个 task 是引用 template 模块进行 /etc/hosts 文件生成。

这里采用绑定 hosts 的方式，下面是 hosts.j2 模板文件：

```
127.0.0.1    localhost localhost.localdomain localhost4 localhost4.
             localhost4
::1         localhost localhost.localdomain localhost6 localhost6.
             localhost6
{% for host in groups['all'] %}
{{ hostvars[host]['inventory_hostname'] }} {{ hostvars[host]
    ['hostname'] }}
{% endfor %}
```

将 Inventory 的 hosts 文件内所有主机信息绑定到所有主机上。到此就是 base 角色所有的 task 操作，主要是完成一些主机名的更改，Iptables 和 Selinux 的关闭，还有一些 yum 源的配置。

我们再来看下 zabbix-serverrole 的 main.yaml，主要是负责安装和配置 zabbix-server 和 MySQL：

```
---
- name: Install Mysql-Server and zabbix-server
  yum: name={{ item }} state=latest
  with_items:
    - mysql-server
    - zabbix-server
    - zabbix-server-mysql
    - zabbix-web-mysql
- name: Init Mysql
  shell: mysql_install_db
- name: Start mysql-server
```

```

    service: name=mysql state=started enabled=yes
- name: Set mysql admin password
    shell: /usr/bin/mysqladmin -u root password 'ansible'
- name: Create Zabbix master databases
    shell: mysql -u root -pansible -e 'create database zabbix_master
        character set utf8 collate utf8_bin;'
- name: Set Zabbix Master databases grant
    shell: mysql -u root -pansible -e 'grant all privileges on
        zabbix_master.* to zabbix@localhost identified by "master";'
- name: Import zabbix initial data (schema.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < schema.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: Import zabbix initial data (images.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < images.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: Import zabbix initial data (data.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < data.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: change PHP timezone
    shell: sed -e 's@;date.timezone =.*@date.timezone = Asia/
        Shanghai@g' /etc/php.ini
- name: Copy /etc/zabbix/zabbix_server.conf files
    template: src=zabbix_server.conf dest=/etc/zabbix/zabbix_server.
        conf owner=root group=root mode=644
- name: Start Zabbix-Server and httpd
    service: name={{ item }} state=started enabled=yes
    with_items:
        - zabbix-server
        - httpd

```

zabbix-server 的 tasks 比较多，因为它涉及数据库的安装以及配置，这里就不一一介绍了，还有 MySQL 没有使用 Ansible 自带的模块进行 MySQL 数据库和用户的管理，建议编写 task 的时候尽量使用 Ansible 自带的模块进行配置管理，不仅仅是方便使用，而且 Ansible 官方的模块对整个状态管理做得很好。

我们再来看 zabbix-proxyrole 的 main.yaml，主要是负责安装和配置 zabbix-proxy 和 MySQL：

```

---
- name: Install Mysql-Server and zabbix-server
    yum: name={{ item }} state=latest

```

```

with_items:
  - mysql-server
  - zabbix-proxy
  - zabbix-proxy-mysql
- name: Init Mysql
  shell: mysql_install_db
- name: Start mysql-server
  service: name=mysql state=started enabled=yes
- name: Set mysql admin password
  shell: /usr/bin/mysqladmin -u root password 'ansible'
- name: Create Zabbix master databases
  shell: mysql -u root -pansible -e 'create database zabbix_proxy
    character set utf8 collate utf8_bin;'
- name: Set Zabbix Master databases grant
  shell: mysql -u root -pansible -e 'grant all privileges on
    zabbix_proxy.* to zabbix@localhost identified by "proxy";'
- name: Import zabbix initial data (schema.sql)
  shell: mysql -u zabbix -pproxy zabbix_proxy < schema.sql
  chdir=/usr/share/doc/zabbix-proxy-mysql-2.4.6/create/
- name: Copy /etc/zabbix/zabbix_proxy.conf files
  template: src=zabbix_proxy.conf dest=/etc/zabbix/zabbix_proxy.
    conf owner=root group=root mode=644
- name: Start Zabbix-Server and httpd
  service: name=zabbix-proxy state=started enabled=yes

```

zabbix-proxy 的 task 流程跟 zabbix-server 很多地方类似，比如数据库的安装以及配置，引用在 zabbix\_proxy.conf 模板中需要制定 zabbix-server 的信息，下面我们简单来看看这个模板文件（只截取有变量引用的行）：

```

Server={{ zabbix_server }}
Hostname={{ hostname }}
ListenIP={{ inventory_hostname }}

```

最后我们来看看 zabbix-agentrole 的 main.yaml 文件：

```

---
- name: Install Zabbix-Agent
  yum: name=zabbix-agent state=latest
- name: Copy /etc/zabbix/zabbix_agentd.conf
  template: src=zabbix_agentd.conf dest=/etc/zabbix/zabbix_agentd.
    conf owner=root group=root mode=644

```



```

    service: name=mysql state=started enabled=yes
- name: Set mysql admin password
    shell: /usr/bin/mysqladmin -u root password 'ansible'
- name: Create Zabbix master databases
    shell: mysql -u root -pansible -e 'create database zabbix_master
        character set utf8 collate utf8_bin;'
- name: Set Zabbix Master databases grant
    shell: mysql -u root -pansible -e 'grant all privileges on
        zabbix_master.* to zabbix@localhost identified by "master";'
- name: Import zabbix initial data (schema.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < schema.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: Import zabbix initial data (images.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < images.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: Import zabbix initial data (data.sql)
    shell: mysql -u zabbix -pmaster zabbix_master < data.sql
        chdir=/usr/share/doc/zabbix-server-mysql-2.4.6/create
- name: change PHP timezone
    shell: sed -e 's@;date.timezone =.*@date.timezone = Asia/
        Shanghai@g' /etc/php.ini
- name: Copy /etc/zabbix/zabbix_server.conf files
    template: src=zabbix_server.conf dest=/etc/zabbix/zabbix_server.
        conf owner=root group=root mode=644
- name: Start Zabbix-Server and httpd
    service: name={{ item }} state=started enabled=yes
    with_items:
        - zabbix-server
        - httpd

```

zabbix-server 的 tasks 比较多，因为它涉及数据库的安装以及配置，这里就不一一介绍了，还有 MySQL 没有使用 Ansible 自带的模块进行 MySQL 数据库和用户的管理，建议编写 task 的时候尽量使用 Ansible 自带的模块进行配置管理，不仅仅是方便使用，而且 Ansible 官方的模块对整个状态管理做得很好。

我们再来看 zabbix-proxyrole 的 main.yaml，主要是负责安装和配置 zabbix-proxy 和 MySQL：

```

---
- name: Install Mysql-Server and zabbix-server
    yum: name={{ item }} state=latest

```

```

with_items:
  - mysql-server
  - zabbix-proxy
  - zabbix-proxy-mysql
- name: Init Mysql
  shell: mysql_install_db
- name: Start mysql-server
  service: name=mysql state=started enabled=yes
- name: Set mysql admin password
  shell: /usr/bin/mysqladmin -u root password 'ansible'
- name: Create Zabbix master databases
  shell: mysql -u root -pansible -e 'create database zabbix_proxy
    character set utf8 collate utf8_bin;'
- name: Set Zabbix Master databases grant
  shell: mysql -u root -pansible -e 'grant all privileges on
    zabbix_proxy.* to zabbix@localhost identified by "proxy";'
- name: Import zabbix initial data (schema.sql)
  shell: mysql -u zabbix -pproxy zabbix_proxy < schema.sql
    chdir=/usr/share/doc/zabbix-proxy-mysql-2.4.6/create/
- name: Copy /etc/zabbix/zabbix_proxy.conf files
  template: src=zabbix_proxy.conf dest=/etc/zabbix/zabbix_proxy.
    conf owner=root group=root mode=644
- name: Start Zabbix-Server and httpd
  service: name=zabbix-proxy state=started enabled=yes

```

zabbix-proxy 的 task 流程跟 zabbix-server 很多地方类似，比如数据库的安装以及配置，引用在 zabbix\_proxy.conf 模板中需要制定 zabbix-server 的信息，下面我们简单来看看这个模板文件（只截取有变量引用的行）：

```

Server={{ zabbix_server }}
Hostname={{ hostname }}
ListenIP={{ inventory_hostname }}

```

最后我们来看看 zabbix-agentrole 的 main.yaml 文件：

```

---
- name: Install Zabbix-Agent
  yum: name=zabbix-agent state=latest
- name: Copy /etc/zabbix/zabbix_agentd.conf
  template: src=zabbix_agentd.conf dest=/etc/zabbix/zabbix_agentd.
    conf owner=root group=root mode=644

```

```
- name: Start zabbix_agnet
  service: name=zabbix-agent state=started enabled=yes
```

zabbix-agent 的 task 最好理解，主要是负责 zabbix-agentd 的安装以及配置文件和服务的管理。这里采用 template 的方式管理 zabbix-agent 的配置文件，在模板中它会引用 zabbix-server 和 zabbix-proxy 的主机信息。我们来看看 zabbix\_agentd.conf 模板文件的内容（只截取有变量引用的行）：

```
Server={{ zabbix_server }},{{ zabbix_proxy }}
ListenIP={{ inventory_hostname }}
ServerActive={{ zabbix_proxy }}:10052
Hostname={{ hostname }}
```

再看整个目录结构以及文件列表：

total 16

```
drwxr-xr-x  4 shencan  staff  136  8 31 14:21 base
drwxr-xr-x  3 shencan  staff  102  9  5 00:08 group_vars
-rw-r--r--  1 shencan  staff  179  9  5 00:10 hosts
-rw-r--r--  1 shencan  staff  275  9  4 23:45 site.yaml
drwxr-xr-x  4 shencan  staff  136  9  5 00:13 zabbix-agent
drwxr-xr-x  4 shencan  staff  136  9  4 22:41 zabbix-proxy
drwxr-xr-x  4 shencan  staff  136  9  4 23:28 zabbix-server
```

#tree

```
|— base
|   |— files
|   |   |— RPM-GPG-KEY-EPEL-6
|   |   |— RPM-GPG-KEY-ZABBIX
|   |   |— epel.repo
|   |   |— zabbix.repo
|   |— tasks
|   |   |— main.yaml
|   |— templates
|   |   |— hosts.j2
|— group_vars
|   |— all
|— hosts
|— site.yaml
|— zabbix-agent
|   |— tasks
|   |   |— main.yaml
```

```

├── templates
│   └── zabbix_agentd.conf
├── zabbix-proxy
│   ├── tasks
│   │   └── main.yaml
│   └── templates
│       └── zabbix_proxy.conf
└── zabbix-server
    ├── tasks
    │   └── main.yaml
    └── templates
        └── zabbix_server.conf

```

14 directories, 15 files

最后来看看 roles 入口文件 site.yaml:

```

---
- hosts: all
  gather_facts: False
  roles:
    - { role: base, tags: base }
    - { role: zabbix-server, when: "'zabbix-server' in group_names", tags: server }
    - { role: zabbix-proxy, when: "'zabbix-proxy' in group_names", tags: proxy }
    - { role: zabbix-agent, tags: agent }

```

## 9.3 安装部署

所有的 roles 文件编写好之后，我们只需要运行以下命令一键完成 zabbix-server、zabbix-prpxy、zabbix-agent 的安装与部署，也可以通过制定 tags 来选择部署：

```

ansible-playbook -i hosts site.yaml
ansible-playbook -i hosts site.yaml --tags base

```

由于输出结果比较多，这里就不展示输出结果信息。接下来我们就通过 Zabbix UI 去继续完成 Zabbix 的安装部署，通过浏览器访问 <http://192.168.1.100/zabbix>，如图 9-1 所示，按照相应步骤继续即可。

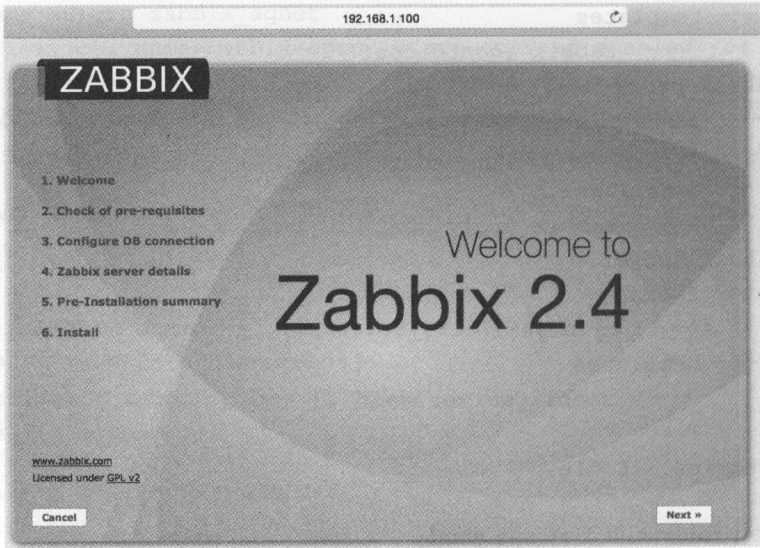


图 9-1 Zabbix 的安装部署

进入数据库后，Zabbix 数据库是 zabbix\_master，用户名是 zabbix，密码是 master，如图 9-2 所示。

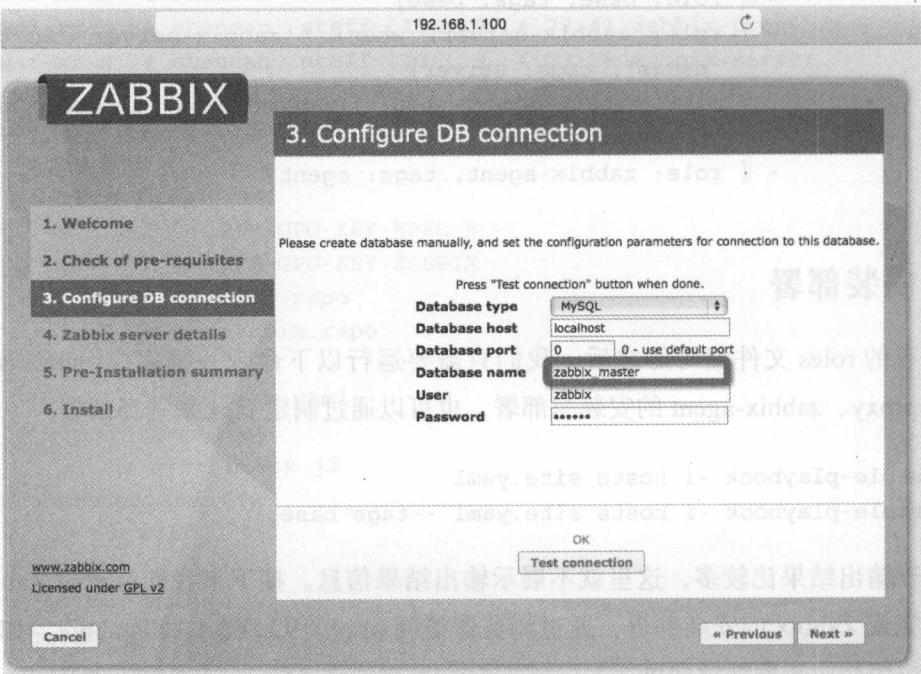


图 9-2 设置数据库



点击 Finish 完成安装，如图 9-3 所示。

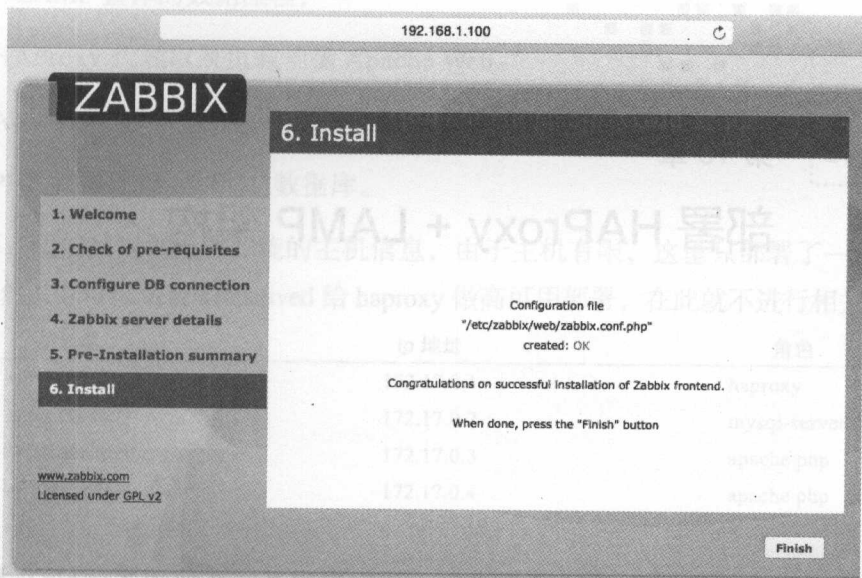


图 9-3 完成安装

使用用户名 Admin 密码 zabbix 即可进入监控首页。

到此为止，我们的 Zabbix 服务端已经安装配置完毕了。关于 proxy 节点的添加，只需通过 Zabbix UI 进行添加即可，在实际的工作中我们会使用 Ansible 部署大量的 Zabbix 客户端程序，所以只需将主机信息添加到 zabbix-agent 组，然后运行 ansible-playbook 即可。关于 Zabbix 客户端主机的添加，在日常生产环境中建议使用 Zabbix API 或者基于 host-metadata 来进行主机自动注册和模板绑定方式进行处理，Ansible 只负责 Zabbix 客户端的安装部署配置即可。更多 Zabbix 的功能可以参考 Zabbix 官网。

## 9.4 本章小结

本章主要介绍如何使用 Ansible 去快速部署 Zabbix 监控系统。部署 Zabbix 监控系统也是本书的第一个实际应用案例，通过这个案例读者会发现 Ansible 自身管理配置功能非常强大而且也很容易上手，正是因为 Ansible 有这些特性使得越来越多的企业都引入 Ansible 作为配置管理工具。在实际的工作中也会经常遇到需要部署一整套跨机器的集群环境，下一章将会介绍如何使用 Ansible 一键部署一个 LAMP 集群环境。

## 部署 HAProxy + LAMP 架构

通过上一章介绍部署 Zabbix 案例之后，我们发现其实在使用 Ansible 做配置管理工作的时候，难点不是如何编写 playbook，而是需要对整个架构的了解，是希望使用 Ansible 去实现什么样子的架构。本章我们继续通过 Ansible 部署一套 HAProxy + LAMP 架构，这个架构也是我们日常运维过程中经常遇到的一个架构，熟悉了一个架构的部署之后，可以非常容易地使用 Ansible 去扩展它。

### 10.1 了解整体架构流程

首先我们知道需要什么样子的架构，然后我们还要了解整个架构每个组件之间是如何衔接和交互的，当然我们还要清楚架构中每个组件的原理和流程。下面我们了解 HAProxy 和 LAMP 集群架构相关的知识。

HAProxy 是一款性能卓越的提供高可用性、负载均衡以及基于 TCP 和 HTTP 应用的代理软件，目前很多大公司也在使用其做 Web 集群和 cache 集群的负载均衡以及代理。

LAMP 架构也是我们每个运维人员熟知的一种网站架构，它是 Linux Apache MySQL 以及 3P (PHP Perl Python) 语言的首字母简称，这种网站架构很容易实现跨主机的横向与纵向扩展，可快速组建一个庞大的 Web 集群系统。

真正的企业级应用中，一般会在 Web 集群系统前面加一层负载均衡系统 (HAProxy

或者 Nginx 或者 LVS)，所以 HAProxy 和 LAMP 集群架构可以无缝结合，下面来了解 HAProxy+LAMP 整体的数据流程：

1) HAProxy 代理以及负载均衡 Apache Web。

2) Apache+PHP 实现 PHP 网站的 Web 功能。

3) PHP 代码连接 MySQL 数据库。

下面，我们来规划部署环境的主机信息，由于主机有限，这里只部署了一台 haproxy 主机。当然大家可以结合 keepalived 给 haproxy 做高可用部署，在此就不进行相关介绍了。

主机名	ip 地址	角色
8f990a898214	172.17.0.1	haproxy
dc6c6f0c8689	172.17.0.2	mysql-server
74596d26bd20	172.17.0.3	apache php
b409dbc12334	172.17.0.4	apache php

## 10.2 编写业务 roles

确定以及规范好主机信息后，我们就可以开始编写 Ansible playbook 了。为了更好地管理以及后续的维护，这里跟上一章一样继续以设备角色为单位编写不同的 roles。我们有 4 个设备角色然后创建 5 个 roles，分别是初始化 base 角色、mysql 角色、haproxy 角色以及 apache 角色。

下面我们来分别介绍每个 roles 的 playbook 以及相关变量和配置模板。首先我们来看初始化 base 角色的 tasks 相关内容：

```
---
- copy: src=CentOS-Base.repo dest=/etc/yum.repos.d/CentOS-Base.repo
  owner=root group=root mode=644
- copy: src=epel.repo dest=/etc/yum.repos.d/epel.repo owner=root
  group=root mode=644
```

因为 base 角色是做主机基础环境相关的配置，所以采用 copy 模块静态管理了两个 yum 源的配置文件。读者可以根据自己主机的情况添加其他的配置管理 tasks。

接下来我们再来看 mysql 角色中的 tasks：

```

- name: Install Mysql-server
  yum: name={{ item }} state=installed
  with_items:
    - mysql-server
    - MySQL-python
- name: Copy my.cnf
  template: src=my.cnf.j2 dest=/etc/my.cnf
  notify:
    - restart mysql
- name: Start Mysql
  service: name=mysql state=started enabled=yes
- name: Create Database
  mysql_db: name={{ database }} state=present
- name: Create Users
  mysql_user: name={{ user }} password={{ password }} priv=*.*:ALL
              host='%' state=present

```

第一个 task 是引用 yum 模块安装 MySQL 服务端相关的包，需要注意的是我这里引用了 Ansible 自带的两个 MySQL 相关的模块，所以前面需要安装 MySQL-python 包。

第二个 task 是采用 Jinja2 模板的方式生成 MySQL 配置文件 my.conf。

第三个 task 是采用 Ansible 自带的 mysql\_db 模块进行 MySQL 数据库的创建。

第四个 task 是采用 Ansible 自带的 mysql\_user 模块进行添加 MySQL 用户以及授权。

接着我们再来看 haproxy 角色的 tasks:

```

---
- name: Install haproxy
  yum: name={{ item }} state=present
  with_items:
    - haproxy
- name: Copy harpoxy.cf
  template: src=haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg owner=
            root group=root mode=644
  notify:
    - restart haproxy
- name: Start haproxy
  service: name=haproxy state=started enabled=yes

```

第一个 task 是引用 yum 模块进行 haproxy 包的安装。

第二个 task 是引用 template 模块进行动态管理 haproxy.cfg 文件，并且触发 handlers。

第三个 task 是引用 service 模块进行 haproxy 服务的管理（运行和开机启动）

最后我们来看一下 apache 角色的 tasks：

```
---
- name: Install Apache and PHP
  yum: name={{ item }} state=present
  with_items:
    - httpd
    - php
    - php-mysql
    - libsemanage-python
    - libselenium-python
- name: Copy index.php.j2
  template: src=index.php.j2 dest=/var/www/html/index.php
- name: http service state
  service: name=httpd state=started enabled=yes
```

因为这里规划的是 Apache 和 PHP 跑在同一台设备上，所以第一个 task 引用 yum 模块进行 Apache 和 PHP 相关的安装。第二个 tasks 是引用 template 模块进行生成一个 PHP 的 HTML 测试页面。第三个 tasks 是引用 service 模块确保 Apache 服务是运行的。

以上几个 roles 的 tasks 相对还是比较简单的，因为我们在上面的 tasks 过程中采用了 template 模板进行动态生成文件，所以我们需要了解相关的变量定义以及配置模板的变量引用。

首先我们来看 Inventory 文件：

```
[apache]
172.17.0.3
172.17.0.4
[mysql]
172.17.0.2
[haproxy]
172.17.0.1
```

这里分为三个业务角色组，每台主机属于不同的业务角色组，下面我们再来看变



量定义文件：

```
[root@yottaa LAMP]# ll group_vars/
总用量 12
-rw-r--r--. 1 root root 29 9月 12 22:58 all
-rw-r--r--. 1 root root 35 9月 12 22:56 haproxy
-rw-r--r--. 1 root root 71 9月 12 22:55 mysql
```

这里分别针对所有主机和主机组进行了相关变量的定义，如下所示：

```
[root@yottaa LAMP]# cat group_vars/all
---
ansible_ssh_pass: 123456
[root@yottaa LAMP]# cat group_vars/haproxy
---
mode: http
balance: roundrobin
[root@yottaa LAMP]# cat group_vars/mysql
---
mysql_port: 3306
user: ansible
password: ansible
database: ansible
```

所定义的变量会在不同的 roles 模板中进行引用。下面我们主要来看一下 apache 测试页面和 haproxy 配置文件的模板内容。首先看 haproxy.cf.j2 文件的内容：

```
global
    log                127.0.0.1 local2
    chroot              /var/lib/haproxy
    pidfile             /var/run/haproxy.pid
    maxconn             4000
    user                root
    group               root
    daemon
```

```
global
    maxconn 100000
    daemon
    nbproc 1
    log 127.0.0.1 local3 info
```

```

defaults
    option http-keep-alive
    maxconn 100000
    mode {{ mode }}
    option httplog
    option dontlognull
    option http-server-close
    option redispatch
    retries 3
    timeout connect 5s
    timeout client 20s
    timeout server 10s

frontend ansible
    bind {{ ansible_default_ipv4.address }}:80
    mode {{ mode }}
    log global
    default_backend apache

backend apache
    option httpchk HEAD / HTTP/1.0
    balance {{ balance }}
    {% for host in groups['apache'] %}
    server {{ hostvars[host].ansible_hostname }} {{ hostvars[host].
        ansible_default_ipv4.address }}:80 check inter 3000 rise 3 fall
        2
    {% endfor %}

```

我们这里主要关心 Jinja2 相关的配置，在 HAProxy 的 apache backend 中会把 Inventory 中 apache 组中的主机都会当作 HAProxy 的后端代理主机。

最后我们来看一下 Apache 和 PHP 的测试页面模板：

```

<?php
    echo "Show Databases List:\n ";
    {% for host in groups['mysql'] %}
    $link = mysqli_connect('{{ hostvars[host].ansible_default_ipv4.
        address }}', '{{ hostvars[host].user }}', '{{ hostvars[host].
        password }}') or die(mysqli_connect_error($link));
    {% endfor %}
    $res = mysqli_query($link, "show databases;");
    while ($row = mysqli_fetch_assoc($res)) {
        echo $row['Database'] . "\n";
    }

```

```
}
?>
```

以上是一个 PHP 页面，没有接触过 PHP 没关系，我们只要关心模板相关的内容，这个模板会通过 Inventory 文件中定义的 MySQL 主机信息和 group\_vars/ 下定义的变量渲染出 PHP 连接 MySQL 主机的信息，最后验证 PHP 能不能正常访问 MySQL 数据库里面的数据。

## 10.3 配置部署以及测试

roles 角色都编写完成，一切准备工作就绪后，我们就可以进行相关的配置确认并进行配置部署和测试工作了。首先我们来看当前工作目录结构：

```
[root@yottaa LAMP]# tree
```

```
├── group_vars
│   ├── all
│   ├── haproxy
│   └── mysql
├── hosts
├── roles
│   ├── apache
│   │   ├── tasks
│   │   │   └── main.yaml
│   │   └── templates
│   │       └── index.php.j2
│   ├── base
│   │   ├── files
│   │   │   ├── CentOS-Base.repo
│   │   │   └── epel.repo
│   │   └── tasks
│   │       └── main.yaml
│   ├── haproxy
│   │   ├── handlers
│   │   │   └── main.yaml
│   │   ├── tasks
│   │   │   └── main.yaml
│   │   └── templates
│   │       └── haproxy.cfg.j2
```

```

├── mysql
│   ├── handlers
│   │   └── main.yaml
│   ├── tasks
│   │   └── main.yaml
│   └── templates
│       └── my.cnf.j2
└── site.yaml

```

16 directories, 16 files

最后编写 roles 的 site.yaml 入口文件:

```

---
- name: Init base environment for all hosts
  hosts: all          # 所有主机引用 base 角色
  roles:
    - base

- name: Install Mysql
  hosts: mysql        #mysql 主机组引用 mysql 角色
  roles:
    - mysql

- name: Install Apache and PHP
  hosts: apache        #apache 主机组引用 apache 角色
  roles:
    - apache

- name: Install Haproxy
  hosts: haproxy       #haproxy 主机组引用 haproxy 角色
  roles:
    - haproxy

```

这里根据不同的主机组引用不同的 roles 进行相关的配置部署。在我们确认语法没问题之后就可以开始进行配置部署了:

```
[root@yottaa LAMP]# ansible-playbook -i hosts site.yaml --syntax-check
```

```
playbook: site.yaml
```

```
[root@yottaa LAMP]# ansible-playbook -i hosts site.yaml
```

```
PLAY [Init base environment for all hosts] *****
```

```
GATHERING FACTS *****
```

```

ok: [172.17.0.3]
ok: [172.17.0.1]
ok: [172.17.0.4]
ok: [172.17.0.2]
----- 此处省略 N 行 -----
PLAY RECAP *****
172.17.0.1      : ok=7    changed=0    unreachable=0    failed=0
172.17.0.2      : ok=9    changed=0    unreachable=0    failed=0
172.17.0.3      : ok=7    changed=0    unreachable=0    failed=0
172.17.0.4      : ok=7    changed=0    unreachable=0    failed=0

```

确认一切 ok 之后就可以进行相关的测试了。下面我们使用 curl 命令直接请求 HAProxy 代理服务器进行测试：

```

[root@yottaa LAMP]# curl 172.17.0.1
LAMP roles Examples:
Show Databases List:
    information_schema
ansible
mysql
test

```

可以看到正常显示 172.17.0.2 数据库里面所有的 databases，说明整个 HAProxy+LAMP 架构已经服务正常，同样也说明 Ansible 的部署也成功了。

## 10.4 扩容与维护

在实际的日常维护工作中难免会遇到需要临时添加或者删除机器。比如上面我们部署的 HAProxy+LAMP 架构，如果哪天发现 Apache 业务有性能瓶颈，我们需要快速扩张整个架构，这个时候我们只需把新添加的机器添加到 hosts 文件中相应的业务角色组下即可。比如我们要扩容添加一台 Apache 机器，将主机信息添加到 hosts 的 Apache 主机组下，然后运行 playbook 的时候只需指定主机运行指定 tags 即可。等待 Apache 业务部署完成后，只需针对 HAProxy 机器运行 haproxy tags 即可将新加入的机器加入服务。当然如果下线服务，也可以按照上面的思路进行设备下线操作。

在我们实际的工作中经常会遇到这种跨主机以及庞大集群的部署需求，通过前面的章节我们也了解到 Ansible 是一个很灵活以及功能强大的配置工具，所以这些功能在



Ansible 很容易实现，我们不需要花太多的时间去寻找如何编写 playbook，我们需要做的是如何去了解整体架构和每个组件的原理以及配置。从以上内容我们可以看到日常使用 Ansible 其实就是把我們心中的架构和功能用 Ansible 实现代码化、状态化而已。

## 10.5 本章小结

本章通过一个简单案例介绍如何利用 Ansible 部署 LAMP 架构，这是 Ansible 在构建集群甚至跨机器部署上面的入门案例。通过本章案例读者可以清晰地了解到如何用 Ansible 在配置部署过程中实现一个业务逻辑架构，这也是我们在实际工作中经常遇到的。随着公司业务的扩张，会有很多需要维护和部署的集群架构，而这些繁复的工作对于 Ansible 来说易如反掌。

## 大数据环境的应用实战

随着云时代的到来，大数据也吸引了越来越多的关注。大数据不在于掌握庞大的数据信息，而在于对这些含有意义的数据进行专业化处理，即在于提高对数据的“加工能力”，通过“加工”实现数据的“增值”。大数据的价值体现在以下方面：

- 对大量消费者提供产品或服务的企业，可以利用大数据进行精准营销。
- 做小而美模式的中长尾企业，可以利用大数据做服务转型。
- 面临互联网压力之下必须转型的传统企业，需要与时俱进充分利用大数据的价值。

大数据与云计算就像一枚硬币的正反面一样密不可分。大数据必然需要采用分布式架构，对海量数据进行分布式数据挖掘，涌现出了大量创新、适用于大数据的技术，包括大规模并行处理（MPP）数据库、数据挖掘电网、分布式文件系统、分布式数据库、云计算平台、互联网和可扩展的存储系统等。实时的大型数据集分析需要像 MapReduce 这样的框架来向数十、数百或甚至数千台主机分配任务，并在容忍时长内有效地处理、输出分析结果，这需要支撑数据处理的海量资源环境能够高度动态自动伸缩调整。

对于这种大规模的应用场景特别需要有自动化运维支撑手段，本章将结合某运营商大数据环境，介绍如何采用 Ansible 进行维护管理。

有些读者可能已经在使用 Hadoop 服务了，甚至也已经做了大量的日常维护操作，对于这些读者阅读本章仍然有助于深入理解如何维护大规模 Hadoop 集群。

## 11.1 某运营商大数据环境

某运营商拥有大量数据资源，有 BSS、OSS、MSS 数据，有 DPI、信令、位置等网络和网管数据，有电信增值业务、行业和公众客户应用数据，有终端、渠道、支付和客服等数据，涵盖参与人、产品、财务、市场营销、事件、地域、资源和账务等类数据。对这几类数据进行加工处理、分析挖掘，对数据进行脱敏处理后，构建数据应用和产品，形成有价值的信息增值。对运营商内部提供面向企业内部的客户行为和消费特征的分析挖掘，实现精确营销、精准维系、效益评价等数据应用业务需求；对合作伙伴通过数据出售、数据咨询、数据能力和数据解决方案四种业务形态实现数据资产的数据运营，最终实现数据资产的增值。

因此统一建设大数据能力产品与应用平台，支持平台、数据对外开放和集约管理，为运营商内外提供大数据服务能力和大数据应用，提供统一能力开发接口、数据产品服务能力封装、数据平台能力封装、资源管理、平台管控等服务，提供 RTB、精准营销、保险反欺诈、选址分析等大数据产品应用。内容主要包括：

- 数据仓储和分析能力：包括基础数据导入、数据访问、数据处理的能力，数据采集和汇总，网络爬虫和规则库管理。
- 数据服务能力组件封装：基于大数据基础数据分析能力，针对不同业务产品特点封装形成数据服务能力组件，包括关键词分析组件、客户识别组件、标签产品组件、行为分析组件等。
- 服务能力统一开放：在数据服务能力组件基础上，对外提供服务能力开放，并对服务过程进行统一管理，包括接入与签权、服务控制、安全管理等。
- 数据管控：包括数据质量管理、数据规则管理、元数据管理等。
- 平台管控与系统管理：对基础计算 / 存储资源管理、服务能力管理、任务和进程调度管理、安全管理、系统管理等。

根据业务需求对整个大数据能力平台进行设计，共分 8 大功能区，包括数据接入区（FTP 服务）、数据汇聚总线（Kafka、FTP）、数据生产加工区（数据存储、计算、封

装)、数据分析区(结构化、非结构化数据)、预上线测试区(中间过程)、服务门户区(产品服务、管理服务、数据接口)、运维服务区(运维管理、ETL 服务、集群应用客户端、堡垒审计),如图 11-1 所示。

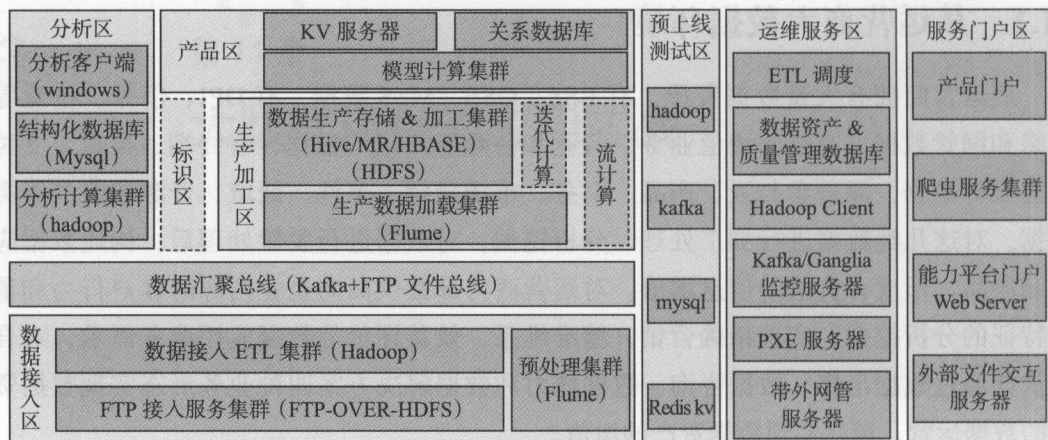


图 11-1 某运营商大数据 8 大功能区

整个大数据平台系统涵盖了数据采集、导入和预处理、统计和分析、数据展现一系列过程。Hadoop 已经是对大数据集进行分布式计算的标准工具,具有完整、充满活力的开源生态环境。构建整个大数据环境涉及部署 Hadoop 集群、HBase、Hive、MySQL、Nginx、Redis、Kafka、Flume、Zookeeper、Storm、Ganglia 等软件。

下面将详细讲解如何用 Ansible 准备 Hadoop 集群基础环境、集群软件部署配置等。

## 11.2 准备大数据集群环境

目前在国内主流的 Hadoop 有 Apache Hadoop (最原始的、所有发行版均基于这个版本进行改进)、Cloudera CDH (Cloudera's Distribution Hadoop)、Hortonworks HDP (Hortonworks Data Platform)。对于初学者或小规模应用大多选择 CDH 版本,与 Apache 发行版本相比,CDH 具有如下优点:

1) CDH 对比 Hadoop 版本的划分非常清晰,当前主要有 CDH4、CDH5,分别对应原生 Hadoop 2.0、Hadoop 2.3。原生的 Apache Hadoop 版本则比较混乱,在兼容性、安全性、稳定性上也需要大量额外的增强。



2) 当前 CDH5 版本是基于 Apache Hadoop 2.3 改进的, 融入了最新的软件补丁, CDH 总能及时跟进最新的 Bug 补丁、新功能组件, 比 Apache Hadoop 同功能版本提早发布, 更新也较快。

3) CDH 增强了安全认证, 支持 Kerberos, Apache Hadoop 只是使用用户名匹配认证。

4) 特别是 CDH 支持 Yum/Apt 包、Tar 包等多种安装方式。适用于各种操作系统, 建议直接用 Yum/Apt 安装, 能够自动匹配安装版本、下载安装依赖软件, 升级非常方便。

下面将详细介绍基于 CDH5 的 Ansible 自动化部署过程, 包括操作系统基础环境、Hadoop 基础环境、Hadoop 集群部署的详细过程。

硬件环境如表 11-1 所示。

表 11-1 硬件环境

序号	主机名	IP 地址	运行服务
1	ah-ansible	172.16.1.220	Ansible, Client
2	ah-namenode	172.16.1.221	Primary NameNode
3	ah-secondary-namenode	172.16.1.222	Secondary NameNode
4	ah-resourcemanager	172.16.1.223	ResourceManager
5	ah-datanode-01	172.16.1.224	DataNode1
6	ah-datanode-02	172.16.1.225	DataNode2
7	ah-datanode-03	172.16.1.226	DataNode3
8	ah-datanode-04	172.16.1.227	DataNode4

软件环境如表 11-2 所示。

表 11-2 软件环境

序号	项目	软件	版本	说明
1	操作系统	CentOS	6.5 x64	国内网站或 <a href="http://www.centos.org">www.centos.org</a>
2	Hadoop	CHD	5.4.7	<a href="http://archive.cloudera.com/cdh5/cdh/5/hadoop-2.6.0-cdh5.4.7.tar.gz">http://archive.cloudera.com/cdh5/cdh/5/hadoop-2.6.0-cdh5.4.7.tar.gz</a>
3	JDK	Java	1.7.0_80	<a href="http://www.oracle.com">www.oracle.com</a> 官网下载
4	自动化工具	Ansible	1.9.2	Yum 安装
5	语言环境	Python	2.6.6	CentOS6.5 自带

参照 Ansible 最佳实践建议, 整个配置管理主要包含在以下 3 个目录:

- group\_vars: 全局定义的变量参数。
- playbooks: 下面又分为: conf (环境配置)、operation (日常维护) 两个目录的脚本。



- roles：包含各个服务的角色，每个角色一般包含 defaults、vars、files、tasks、handlers 等目录。

playbooks 文件见表 11-3。

表 11-3    playbooks 文件说明

playbooks 文件	说明
hosts	资源清单文件，包含一系列主机组
ansible_cobbler.yml	创建网络安装操作的 kickstarts 服务
ansible_downloads.yml	软件下载脚本
ansible_prepare.yml	HadoopLinux 环境初始化准备脚本，包括创建账号、无口令密钥登录、修改内核参数、修改文件句柄等
ansible_hadoop.yml	完整的安装 CDH5 的安装部署脚本
ansible_post.yml	安装完成 CDH5 之后的后续初始化 Hadoop 脚本，调用下面的 post_install_setups 角色

目录说明如表 11-4 所示。

表 11-4    目录说明

目录	说明
file_archives	存放 CDH 源码安装软件包 hadoop-2.6.0-cdh5.4.7.tar.gz，jdk-7u80-linux-x64.tar.gz
group_vars	该目录下包含定义 Hadoop 环境设置的全局变量的文件 all，all 中定义了安装目录、hadoop 用户、storm 用户、软件版本、dhfs 参数、yarn 参数、map_reduce 参数、java_home 目录等参数
roles	该目录下包含各个 role 的功能组件
commons	负责定义、处理 Linux 操作系统的基本参数
cobbler	部署 cobbler 服务，支持网络安装操作系统
jdk	JDK 安装部署
ssh_known_hosts	处理 SSH 第一次登录时，提示加入 known_host
ssh_password_less	配置 ssh 无密码、密钥登录方式
cdh5_commons	安装、配置操作系统基本环境
cdh5_namenode_primary	安装、部署 Hadoop 的 NameNode 主节点服务
cdh5_namenode_secondary	安装、部署 Hadoop 的 NameNode 辅节点服务
cdh5_resourcemgr	安装、部署 Hadoop 的资源管理节点服务
cdh5_datanode	安装、部署 Hadoop 的 DataNode 节点服务
post_install_setups	创建 Hadoop 用户、HDFS 目录、修改目录权限等

如果有多个 Hadoop 集群环境，如有生产集群、测试集群等，可以分别定义，分开管理。也可以定义多个资源清单文件来表示各个集群环境，这样也方便由 ansible-playbook 指定不同的集群环境进行操作。

### 11.2.1 安装操作系统

对于 Hadoop 环境，有大量的主机需要初始化安装操作系统，Cobbler 是无需进行人工干预即可安装操作系统的理想选择。Cobbler 设置一个 PXE 引导环境，并控制与安装相关的所有方面。Cobbler 功能如下：

- 使用一个以前定义的模板来配置 DHCP 服务（如果启用了管理 DHCP）。
- 将在一个存储库（yum 或 rsync）建立镜像，以注册一个新操作系统。
- 在 DHCP 配置文件中为需要安装的服务器创建一个条目，并使用您指定的参数（IP 和 MAC 地址）。
- 在 TFTP 服务目录下创建适当的 PXE 文件。
- 重新启动 DHCP 服务以反映更改。

Cobbler 支持众多的发行版：Red Hat、Fedora、CentOS、Debian、Ubuntu 和 SuSE。下面介绍 Ansible 如何编写自动化脚本来部署一个完整的 Cobbler 环境，主要脚本及其实现功能见表 11-5。

表 11-5 Ansible 主要脚本及其功能

文件及目录	说明
hosts	Cobbler 服务部署的资源清单文件
ansible_cobbler.yml	ansible_cobbler.yml 是 playbook 执行的入口文件，它将引用变量定义文件 group_vars/cobbler.yml，执行 cobbler/tasks/main.yml，并根据处理情况调用相应的处理器 common/handlers/cobbler.yml 和 cobbler/handlers/main.yml
cobbler/files/var/lib/cobbler/kickstarts/ cluster.ks	Kickstarts 定义需要安装主机的操作系统的参数
cobbler/handlers/main.yml	处理 Cobbler 配置文件同步，并处理因 Cobbler 配置变更需要重启 cobblerd、httpd、xinetd 等服务
cobbler/tasks/main.yml	安装 Cobbler 软件； 部署 tftp、cobbler、kickstarts、dhcp 服务的模板，并启动服务
cobbler/templates/etc/cobbler/dhcp.template	dhcp 配置模板文件
cobbler/templates/etc/cobbler/modules.conf	加载模块 authentication、authorization、dns、dhcp、tftpd
cobbler/templates/etc/cobbler/settings	Cobbler 环境设置参数文件
common/handlers/cobbler.yml	配置文件变更之后，处理重启服务，重启主机
group_vars/cobbler.yml	Cobbler 部署主机 IP 地址范围参数 cobbler_temporary_range: 172.16.1.11 ~ 172.16.1.254

#### 1. Cobbler 资源清单文件

在资源清单文件 hosts 中配置 [cobbler] 主机组，下面包含需要部署 Cobbler 的主机：

```
[cobbler]
```

```
cobbler
```

## 2. cobbler.yml

是 playbook 程序执行的入口，将会调用 `cobbler/tasks/main.yml` 文件：

```
---
- hosts: cobbler
  user: root
  sudo: yes
  vars_files:
    - group_vars/cobbler.yml
  tasks:
    - include: cobbler/tasks/main.yml
  handlers:
    - include: common/handlers/cobbler.yml
    - include: cobbler/handlers/main.yml
```

## 3. cobbler/tasks/main.yml

该脚本包含 Cobbler 及其依赖软件包的安装，然后启动 `ftpt`、分发 Cobbler 模板文件、拷贝 `kickstarts` 模板文件，最后确保 `ftpt`、`dhcp`、`obblerd`、`xinetd`、`httpd` 等服务在运行着：

```
---
- name: Install Cobbler
  yum: name=$item state=installed
  with_items:
    - cobbler
    - cobbler-web
    - xinetd
    - tftpd
    - pykickstart
    - httpd
    - dhcp
    - syslinux

- name: Enable tftpd
  lineinfile: dest=/etc/xinetd.d/tftpd regexp='^disable=' line=disable=no
  notify: Restart service xinetd
  tags:
```

```

- services
- xinetd

- name: Cobbler templates
  template: src=cobbler/templates/etc/cobbler/$item dest=/etc/cobbler/
  owner=root group=root
  with_items:
  - dhcp.template
  - modules.conf
  - settings
  notify:
  - Sync cobbler
  - Restart service cobblerd
  tags:
  - templates

- name: Cobbler kickstart script
  copy: src=cobbler/files/var/lib/cobbler/kickstarts/cluster.ks dest=
    /var/lib/cobbler/kickstarts/ owner=root group=root

- name: Enable and Start Services
  service: name=$item state=started enabled=yes
  with_items:
  - cobblerd
  - xinetd
  - httpd
  tags:
  - services

- name: Enable dhcpd Services
  service: name=dhcpd enabled=yes
  notify:
  - Sync cobbler
  - Restart service cobblerd
  tags:
  - services

```

#### 4. var/cobbler.yml

定义 Cobbler 部署主机 IP 地址范围参数 `cobbler_temporary_range`，本例中为 172.16.1.11 ~ 172.16.1.254。

```
cobbler_temporary_range: 172.16.1.11 ~ 172.16.1.254
```

### 11.2.2 操作系统初始化

要部署 Hadoop，首先需要对操作系统进行初始化，每台主机节点需要配置：

- 主机节点的主机名、IP 地址信息需要在所有主机节点的 `/etc/hosts` 中保持一致。
- 关闭 SELinux 服务。
- 关闭 IPv6 选项。
- 配置 Linux 内核参数，包括句柄数、TCP 参数。

上述基础环境需要对每台主机都进行配置，对于大型的 Hadoop 集群架构通过手工命令操作是很费时费力的任务。而 Ansible 自动化可以很容易完成这些任务。

我们做了 8 台服务器组的实例，但很容易扩展到数百台的 Hadoop 集群环境。每台主机节点安装 CentOS 6.5 x64 操作系统，并安装了 Python 和 `libselinux-python` 安装包。即使关闭 `selinux` 服务，`libselinux-python` 的软件包也是需要安装的，当然这也可以通过 Ansible 进行安装。

初始化 Linux 操作系统基础环境，我们单独编写了个角色 `commons`，在 `commons` 角色中包含 3 个目录 `defaults`、`templates`、`task` 分别对应配置参数值设置、配置文件模板、playbook 执行脚本，具体如表 11-6 所示。

表 11-6 执行脚本说明

文件	说明
<code>roles/commons/tasks/main.yml</code>	初始化 Linux 操作系统的 playbook 脚本文件
<code>roles/commons/defaults/main.yml</code>	定义基础环境参数值
<code>roles/commons/templates/90-nproc.conf</code>	设置操作系统句柄数配置模板
<code>roles/commons/templates/etc_hosts</code>	<code>/etc/hosts</code> 文件模板
<code>roles/commons/templates/hdadmin.conf</code>	Hadoop 管理用户 <code>hdadmin</code> 参数设置模板
<code>roles/commons/templates/limits.conf</code>	文件最大打开数模板
<code>roles/commons/templates/sysctl.conf</code>	Linux 系统内核参数模板

#### 1. 定义环境变量参数

这些参数主要是配置系统句柄数、内存参数、TCP 参数，将在模板文件中被引用，详见下面：

```
$cat ~/roles/commons/defaults/main.yml
```

```
nproc_conf:
```



```

all_user_soft_limit: 10240
all_user_hard_limit: 10240
root_user_soft_limit: unlimited
# hdadmin nproc parameters.
hdadmin_soft_nofile: 32768
hdadmin_soft_nproc: 65536
hdadmin_hard_nofile: 1048576
hdadmin_hard_nproc: unlimited
hdadmin_hard_memlock: unlimited

#
# limits.conf variables
#
limits_conf:
    all_user_soft_limit: 1000001
    all_user_hard_limit: 1000001

#
# sysctl.conf variables
#
sysctl_conf:
    # vm setting
    vm_swappiness: 10
    vm_dirty_ratio: 10
    vm_min_free_kbytes: 65536
    vm_max_map_count: 262144
    # kernel parameter
    kernel_msgmnb: 100000
    kernel_msgmax: 100000
    # filesystem
    fs_file-max: 2097152
    # net core parameters
    net_core_rmem_default: 1048576
    net_core_rmem_max: 16777216
    net_core_wmem_default: 1048576
    net_core_wmem_max: 16777216
    net_core_optmem_max: 25165824
    net_core_somaxconn: 65536
    net_core_netdev_max_backlog: 65536
    # ipv4 Setting
    net_ipv4_tcp_moderate_rcvbuf: 0
    net_ipv4_conf_all_rp_filter: 1

```

```

net_ipv4_tcp_slow_start_after_idle: 0
net_ipv4_tcp_fin_timeout: 10
net_ipv4_tcp_ecn: 0
net_ipv4_tcp_max_syn_backlog: 100000
net_ipv4_tcp_max_orphans: 262144
net_ipv4_tcp_max_tw_buckets: 2000000
net_ipv4_tcp_sack: 1
net_ipv4_tcp_timestamps: 1
net_ipv4_tcp_fin_timeout: 10
net_ipv4_tcp_slow_start_after_idle: 0
# Congestion Algo to use.
net_ipv4_tcp_congestion_control: cubic
# tcp memory
net_ipv4_tcp_mem_min: 30000000
net_ipv4_tcp_mem_default: 30000000
net_ipv4_tcp_mem_max: 30000000
net_ipv4_tcp_rmem_min: 30000000
net_ipv4_tcp_rmem_default: 30000000
net_ipv4_tcp_rmem_max: 30000000
net_ipv4_tcp_wmem_min: 30000000
net_ipv4_tcp_wmem_default: 30000000
net_ipv4_tcp_wmem_max: 30000000
#
net_ipv4_tcp_tw_reuse: 1
net_ipv4_tcp_tw_recycle: 1
net_unix_max_dgramqlen: 100
net_ipv4_ip_nonlocal_bind: 1
net_ipv4_tcp_synack_retries: 2
net_ipv4_tcp_syn_retries: 2

```

## 2. 定义模板文件

在 Linux 环境中部署 Hadoop，由于 Hadoop 有大量的连接请求，而 Linux 是有文件句柄限制的，默认配置不是很高，一般是 1024，生产服务器很容易就达到这个数量，这将严重影响服务器的最大并发数：

```
$cat ~/roles/commons/templates/90-nproc.conf
```

```

*      soft      nproc      {{ nproc_conf['all_user_soft_limit'] }}
*      hard      nproc      {{ nproc_conf['all_user_hard_limit'] }}
root   soft      nproc      {{ nproc_conf['root_user_soft_limit'] }}

```

配置每台主机的 /etc/hosts 文件，将根据 Ansible 的清单文件中配置的 hosts 生成，

将分发到定义的所有主机节点，是所有节点的名字解析都一样。

```
$cat ~/roles/commons/templates/etc_hosts
```

```
127.0.0.1    localhost
{% for host in groups['allnodes'] %}
{{ hostvars[host]['inventory_hostname'] }} {{ hostvars[host].host_name }}
{% endfor %}
```

设置 hdadmin 用户的句柄参数：

```
$cat ~/roles/commons/templates/hdadmin.conf
```

```
# Current below User -> 10240 and root -> max limit.
{{ hadoop_user }} soft nfile {{ nproc_conf['hdadmin_soft_nfile'] }}
{{ hadoop_user }} soft nproc {{ nproc_conf['hdadmin_soft_nproc'] }}
{{ hadoop_user }} hard nfile {{ nproc_conf['hdadmin_hard_nfile'] }}
{{ hadoop_user }} hard nproc {{ nproc_conf['hdadmin_hard_nproc'] }}
{{ hadoop_user }} hard memlock {{ nproc_conf['hdadmin_hard_memlock'] }}
```

设置最大文件打开数：

```
$cat ~/roles/commons/templates/limits.conf
```

```
# - nfile - max number of open file descriptors
* soft nfile {{ limits_conf['all_user_soft_limit'] }}
* hard nfile {{ limits_conf['all_user_hard_limit'] }}
```

设置主机节点的内核参数、内存参数、网络连接参数：

```
roles/commons/templates/sysctl.conf
```

```
net.ipv4.ip_forward = 0
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.default.accept_source_route = 0
kernel.sysrq = 0
kernel.core_uses_pid = 1
net.ipv4.tcp_syncookies = 0
kernel.shmmax = 68719476736
kernel.shmall = 4294967296
```

```
vm.dirty_ratio = {{ sysctl_conf['vm_dirty_ratio'] }}
```

```
vm.swappiness = {{ sysctl_conf['vm_swappiness'] }}
```

```

kernel.msgmnb = {{ sysctl_conf['kernel_msgmnb'] }}
kernel.msgmax = {{ sysctl_conf['kernel_msgmax'] }}
fs.file-max = {{ sysctl_conf['fs_file-max'] }}
net.core.rmem_default = {{ sysctl_conf['net_core_rmem_default'] }}
net.core.rmem_max = {{ sysctl_conf['net_core_rmem_max'] }}
net.core.wmem_default = {{ sysctl_conf['net_core_wmem_default'] }}
net.core.wmem_max = {{ sysctl_conf['net_core_wmem_max'] }}
net.core.optmem_max = {{ sysctl_conf['net_core_optmem_max'] }}
net.core.somaxconn = {{ sysctl_conf['net_core_somaxconn'] }}
net.core.netdev_max_backlog = {{ sysctl_conf['net_core_netdev_max_
    backlog'] }}
net.ipv4.tcp_moderate_rcvbuf = {{ sysctl_conf['net_ipv4_tcp_moderate_
    rcvbuf'] }}
net.ipv4.conf.all.rp_filter = {{ sysctl_conf['net_ipv4_conf_all_rp_
    filter'] }}
net.ipv4.ip_local_port_range = 4096 65535
net.ipv4.tcp_congestion_control = {{ sysctl_conf['net_ipv4_tcp_
    congestion_control'] }}
net.ipv4.tcp_ecn = {{ sysctl_conf['net_ipv4_tcp_ecn'] }}
net.ipv4.tcp_max_syn_backlog = {{ sysctl_conf['net_ipv4_tcp_max_syn_
    backlog'] }}
net.ipv4.tcp_max_orphans = {{ sysctl_conf['net_ipv4_tcp_max_orphans'] }}
net.ipv4.tcp_max_tw_buckets = {{ sysctl_conf['net_ipv4_tcp_max_tw_
    buckets'] }}
net.ipv4.tcp_sack = {{ sysctl_conf['net_ipv4_tcp_sack'] }}
net.ipv4.tcp_timestamps = {{ sysctl_conf['net_ipv4_tcp_timestamps'] }}
net.ipv4.tcp_fin_timeout = {{ sysctl_conf['net_ipv4_tcp_fin_timeout'] }}
net.ipv4.tcp_slow_start_after_idle = {{ sysctl_conf['net_ipv4_tcp_slow_
    start_after_idle'] }}
net.ipv4.tcp_mem = {{ sysctl_conf['net_ipv4_tcp_mem_min'] }} {{ sysctl_
    conf['net_ipv4_tcp_mem_default'] }} {{ sysctl_conf['net_ipv4_tcp_
    mem_max'] }}
net.ipv4.tcp_rmem = {{ sysctl_conf['net_ipv4_tcp_rmem_min'] }} {{ {
    sysctl_conf['net_ipv4_tcp_rmem_default'] }} {{ sysctl_conf['net_
    ipv4_tcp_rmem_max'] }}
net.ipv4.udp_rmem_min = 16384
net.ipv4.tcp_wmem = {{ sysctl_conf['net_ipv4_tcp_wmem_min'] }} {{ {
    sysctl_conf['net_ipv4_tcp_wmem_default'] }} {{ sysctl_conf['net_
    ipv4_tcp_wmem_max'] }}
vm.max_map_count = {{ sysctl_conf['vm_max_map_count'] }}
net.ipv4.tcp_tw_reuse = {{ sysctl_conf['net_ipv4_tcp_tw_reuse'] }}
net.ipv4.tcp_tw_recycle = {{ sysctl_conf['net_ipv4_tcp_tw_recycle'] }}

```

```

net.unix.max_dgram_qlen = {{ sysctl_conf['net_unix_max_dgram_qlen'] }}
net.ipv4.ip_nonlocal_bind = {{ sysctl_conf['net_ipv4_ip_nonlocal_bind']
}}
net.ipv4.tcp_synack_retries = {{ sysctl_conf['net_ipv4_tcp_synack_
retries'] }}
net.ipv4.tcp_syn_retries = {{ sysctl_conf['net_ipv4_tcp_syn_retries'] }}
vm.min_free_kbytes = {{ sysctl_conf['vm_min_free_kbytes'] }}

```

这些模板中的变量值已经在前面的参数默认值进行了设置，这里只是引用。

### 3. 定义执行部署的 playbook。

把这些模板拷贝到目标节点上，然后进行配置：

```
$cat ~/roles/commons/tasks/main.yml
```

```

---
- name: Update sysctl.conf on the server.
  template: src=sysctl.conf dest=/etc/sysctl.conf owner=root
            group=root backup=yes mode=0600

- name: Update limits.conf on the server.
  template: src=limits.conf dest=/etc/security/limits.conf
            backup=yes mode=0644

- name: Update '90-nproc.conf' on the server.
  template: src="90-nproc.conf" dest=/etc/security/limits.d/
            owner=root group=root backup=yes mode=0644

- name: Update 'etc/hosts' on the server.
  template: src=etc_hosts dest=/etc/hosts owner=root group=root
            backup=yes mode=0644

- name: Update `{{ hadoop_user }}.conf` on the server.
  template: src=hadoop.conf dest=/etc/security/limits.d/
            {{ hadoop_user }}.conf owner=root group=root backup=yes
            mode=0644

- name: Setting `sysctl.conf` configuration.
  command: sysctl -p

- name: Update Hostname (/etc/sysconfig/network)

```



```

lineinfile: dest=/etc/sysconfig/network regexp='^HOSTNAME'
            line="HOSTNAME={{ host_name }}" state=present

- name: Setting the Hostname Without a restart.
  command: hostname {{ host_name }}

```

上述 playbook 文件中包含了一系列的处理步骤。为了让 selinux 配置生效，需要把所有的主机都重启一遍。Ansible 具有幂等特性，也就是多次执行同样的 playbook，只要主机节点已经配置好就不会对配置内容进行修改。这个特性可以用于检查配置项是否发生变化，或者增加新的主机节点到主机组中。

### 11.2.3 Ansible 无口令密钥执行环境

在配置 Ansible 无口令密钥执行环境时，请先检查 epel-release、libselinux-python、sshpass 这三个软件包确保已经安装，然后按照如下步骤操作。

#### 1. known\_hosts 中添加远程节点的公钥信息

为了不在 SSH 第一次登录时候出现添加到 known\_hosts 的提示，先把公钥信息附加到 /etc/ssh/ssh\_known\_hosts 文件中：

```
$cat ~/role/ssh_known_hosts/tasks/main.yml
```

```

---
- name: Make sure the known hosts file exists
  file: "path={{ ssh_known_hosts_file }} state=touch"

- name: Check host name availability
  shell: "ssh-keygen -f {{ ssh_known_hosts_file }} -F {{ item }}"
  with_items: groups['sshknownhosts']
  register: ssh_known_host_results

- name: Scan the public key
  shell: "{{ ssh_known_hosts_command }} {{ item.item }} >> {{ ssh_
        known_hosts_file }}"
  with_items: ssh_known_host_results.results
  when: item.stdout == ""

```

#### 2. 无密码 SSH 远程登录

把生产的密钥对放在 role/ssh\_password\_less/templates/ 目录下，包括 id\_rsa、id\_

rsa.pub。然后在 tasks 目录下添加 playbook 执行脚本：

```
$cat ~/role/ssh_password_less/tasks/main.yml
```

```
---
- name: Create a User "{{{ hadoop_user }}}" for all our Hadoop
  Modules.
  user: name={{{ hadoop_user }}} password={{{ hadoop_password }}}
- name: Create a .ssh Directory.
  file: path=~/.ssh state=directory owner={{{ hadoop_user }}}
      group={{{ hadoop_group }}} mode=0700
  sudo: yes
  sudo_user: "{{{ hadoop_user }}}"
- name: Lets copy the template id_rsa to auth_keys location.
  template: src=id_rsa.pub dest=~/.ssh/authorized_keys mode=644
  sudo: yes
  sudo_user: "{{{ hadoop_user }}}"
- name: Lets copy id_rsa to location .ssh.
  template: src=id_rsa dest=~/.ssh/id_rsa mode=600
  sudo: yes
  sudo_user: "{{{ hadoop_user }}}"
- name: Lets copy id_rsa.pub to location .ssh.
  template: src=id_rsa.pub dest=~/.ssh/id_rsa.pub mode=644
  sudo: yes
  sudo_user: "{{{ hadoop_user }}}"
```

注：模板中的配置参数是定义在 group\_vars 目录下。

### 11.2.4 安装、配置 JDK

把下载的 Hadoop 软件、Java 源码安装包复制到 file\_archives 目录。在本例中将用到两个主要源码文件：hadoop-2.6.0-cdh5.4.7.tar.gz（CDH5 源码安装包）和 jdk-7u80-linux-x64.tar.gz（Java 源码安装包）。

对于安装 Java 构成比较简单，首先对 Java 的软件源、软件版本进行定义，然后创建目录、安装解压、设置环境变量、创建链接等就完成了。

1) 定义 JDK 部署的参数。

```
$cat ~/roles/jdk/defaults/main.yml
```

```
jdk_download_filename: jdk-7u80-linux-x64.tar.gz
jdk_version: jdk1.7.0_80
```

## 2) JDK 部署脚本。创建 Java 安装目录、设置环境变量、创建软件链接：

```
$cat ~/roles/jdk/tasks/main.yml
```

```
- name: JDK | Make sure openjdk is uninstalled
  yum: pkg=openjdk state=absent

- name: JDK | Make a directory that holds the Java binaries
  file: path=/usr/local/java state=directory

- name: JDK | Unarchive Oracle JDK
  unarchive: src=file_archives/{{ jdk_download_filename }} dest=/usr/
    local/java chdir=/usr/local/java creates=/usr/local/java/{{ jdk_
    version }}

- name: JDK | Update the symbolic link to the JDK install
  file: path={{ java_home }} src=/usr/local/java/{{ jdk_version }}
    state=link force=yes

- name: JDK | Add the JDK binaries to the system path (/etc/profile)
  lineinfile: dest=/etc/profile regexp='^JAVA_HOME={{ java_home }}'
    line="JAVA_HOME={{ java_home }}" state=present

- name: JDK | Add the JDK binaries to the system path (/etc/profile)
  lineinfile: dest=/etc/profile regexp='^PATH=.*JAVA_HOME.*'
    line="PATH=$PATH:$HOME/bin:$JAVA_HOME/bin" state=present

- name: Remove alternatives before we set them.
  command: rm -f /var/lib/alternatives/{{ item }}
  with_items:
    - java
    - javac
    - javaws
    - javah
    - jar
    - jps

- name: JDK | Inform the system where Oracle JDK is located
  alternatives: name={{ item }} link=/usr/bin/{{ item }} path=/usr/
    local/java/jdk/bin/{{ item }}
  with_items:
    - java
    - javac
    - javaws
    - javah
    - jar
    - jps
```

## 11.3 部署 Hadoop 集群

在 Ansible 自动化管理中,首先需要分析被管节点的功能、需要部署软件、使用的配置文件,根据节点配置参数相同、相似、可继承等方式对节点进行分组,形成 Ansible 的资源清单(inventory),再由 ansible-playbook 对这些分组进行模板、任务的组织。

在本实例中,资源清单直接在 hosts 文件中定义,该文件中包含最顶层的 sshknown hosts、allnodes 主机组,sshknownhosts 主机组包含 hadoopcluster 主机组,然后是 hadoopcluster 包含 namenodes、secondarynamenode、resourcemanager、jobhistoryserver、datanodes 等主机组。

下面是 hosts 文件清单:

```
# 需要初始化 Linux 环境的所有节点
```

```
[allnodes]
```

```
172.16.1.220 host_name=ah-ansible
172.16.1.221 host_name=ah-namenode
172.16.1.222 host_name=ah-secondary-namenode
172.16.1.223 host_name=ah-resourcemanager
172.16.1.224 host_name=ah-datanode-01
172.16.1.225 host_name=ah-datanode-02
172.16.1.226 host_name=ah-datanode-03
172.16.1.227 host_name=ah-datanode-04
```

```
# hadoop cluster
```

```
[namenodes]
```

```
172.16.1.221
```

```
[secondarynamenode]
```

```
172.16.1.222
```

```
[resourcemanager]
```

```
172.16.1.223
```

```
[jobhistoryserver]
```

```
172.16.1.223
```

```
[datanodes]
```

```
172.16.1.224
```

```
172.16.1.225
```

```
172.16.1.226
```

```
172.16.1.227
```

```
[hadoopcluster:children]
```

```
namenodes
```

```
secondarynamenode
```

```
resourcemanager
```

```
jobhistoryserver
```

```
datanodes
```

```
# sshknown hosts list.
```

```
[sshknownhosts:children]
```

```
hadoopcluster
```

在编写 Ansible 自动化脚本时，通常把 playbook 脚本封装成角色（role），便于管理、重用。在本例中我们把 Hadoop 部署的 playbook 脚本封装成多个角色，然后通过 `ansible_hadoop.yml` 将调用 `ssh_password_less`、`cdh5_commons` 角色，初始化 `hadoopcluster` 集群所有节点。然后再对 `namenodes`、`secondarynamenode`、`resourcemanager`、`datanodes` 主机组分别再调用 `cdh5_namenode_active`、`cdh5_namenode_secondary`、`cdh5_resourcemgr`、`cdh5_datanode` 角色进行部署。

部署 Hadoop 的 playbook 脚本 `ansible_hadoop.yml` 将用 `root` 在 `hadoopcluster` 范围内部署，首先调用 `cdh5_commons` 角色对所有主机 Linux 进行初始化，然后对其他不同功能的主机组进行分别部署：

```
ansible_hadoop.yml
```

```
- hosts: hadoopcluster
```

```
  remote_user: root
```

```
  roles:
```

```
    - ssh_password_less
```

```
    - cdh5_commons
```

```
- hosts: namenodes
```

```
  remote_user: root
```

```
  roles:
```

```
    - cdh5_namenode_active
```

```
- hosts: secondarynamenode
```



```

remote_user: root
roles:
  - cdh5_namenode_secondary

- hosts: resourcemanager
  remote_user: root
  roles:
    - cdh5_resourcemgr

- hosts: datanodes
  remote_user: root
  roles:
    - cdh5_datanode

```

下面将对每个角色的部署内容进行讲述。

### 11.3.1 准备 Hadoop 基础角色

准备 Hadoop 角色在 roles/cdh5\_commons 目录下，主要包含 Hadoop 环境配置的模板文件、task 任务执行的 playbook 脚本。

#### 1. 模板文件

##### (1) roles/cdh5\_commons/templates/core-site.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://{% for server in groups['namenodes'] %}{% if not
      loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
      endif %}}{ server }}{% endfor %}:9000</value>
  </property>
</configuration>

```

##### (2) roles/cdh5\_commons/templates/DatanodeScripts.sh

```

# Update /etc/hosts

sudo cp /etc/hosts /etc/hosts.bkpz

```

```

sudo cp etc_new_hosts /etc/hosts

# Create Directories for NN/DN/JN

sudo mkdir -p /data1/nn /data1/jn

for item in 1 2 3 4 5 6;
do
    sudo mkdir -p /data${item}/dn;
    sudo mkdir -p /data${item}/yarn/local;
    sudo mkdir -p /data${item}/yarn/logs;

    sudo chown hdadmin:hdadmin -R /data${item}/dn
    sudo chown hdadmin:hdadmin -R /data${item}/yarn

done;

# Change Directory Permissions.
sudo chown hdadmin:hdadmin /data1/nn
sudo chown hdadmin:hdadmin /data1/jn

```

### (3) roles/cdh5\_commons/templates/etc\_hosts.txt

```

# ETC HOST FILE

172.16.1.220    AH-ANSIBLE      # Service Running on Each Node.
172.16.1.221    AH-NAMENODE        #ANSIBLE / Client
172.16.1.222    AH -STANDBY-NN     # NAMENODE / JOURNALNODE
172.16.1.223    AH -RES-MANAGER    # STANDBY NAMENODE / JOURNALNODE
172.16.1.224    AH -DATANODE-01     # RESOURCE MANAGER / JOURNALNODE
172.16.1.225    AH -DATANODE-02     # DATANODE / NODEMANAGER
172.16.1.226    AH -DATANODE-03     # DATANODE / NODEMANAGER
172.16.1.227    AH -DATANODE-04     # DATANODE / NODEMANAGER

```

### (4) roles/cdh5\_commons/templates/hadoop-env.sh

```

export JAVA_HOME={{ java_home }}

# The jsvc implementation to use. Jsvc is required to run secure
  datanodes.
#export JSVC_HOME=${JSVC_HOME}

```

```

export HADOOP_CONF_DIR=${HADOOP_CONF_DIR:-"/etc/hadoop"}

# Extra Java CLASSPATH elements. Automatically insert capacity-
scheduler.
for f in $HADOOP_HOME/contrib/capacity-scheduler/*.jar; do
if [ "$HADOOP_CLASSPATH" ]; then
    export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$f
else
    export HADOOP_CLASSPATH=$f
fi
done

# The maximum amount of heap to use, in MB. Default is 1000.
#export HADOOP_HEAPSIZE=
#export HADOOP_NAMENODE_INIT_HEAPSIZE=""

# Extra Java runtime options. Empty by default.
export HADOOP_OPTS="$HADOOP_OPTS -Djava.net.preferIPv4Stack=true"

# Command specific options appended to HADOOP_OPTS when specified
export HADOOP_NAMENODE_OPTS="-Dhadoop.security.logger=${HADOOP_SECURITY_
    LOGGER:-INFO,RFAS} -Dhdfs.audit.logger=${HDFS_AUDIT_LOGGER:-
    INFO,NullAppender} $HADOOP_NAMENODE_OPTS"
export HADOOP_DATANODE_OPTS="-Dhadoop.security.logger=ERROR,RFAS
    $HADOOP_DATANODE_OPTS"

export HADOOP_SECONDARYNAMENODE_OPTS="-Dhadoop.security.logger=${HADOOP_
    SECURITY_LOGGER:-INFO,RFAS} -Dhdfs.audit.logger=${HDFS_AUDIT_
    LOGGER:-INFO,NullAppender} $HADOOP_SECONDARYNAMENODE_OPTS"

export HADOOP_NFS3_OPTS="$HADOOP_NFS3_OPTS"
export HADOOP_PORTMAP_OPTS="-Xmx512m $HADOOP_PORTMAP_OPTS"
# The following applies to multiple commands (fs, dfs, fsck, distcp etc)
export HADOOP_CLIENT_OPTS="-Xmx512m $HADOOP_CLIENT_OPTS"
#HADOOP_JAVA_PLATFORM_OPTS="-XX:-UsePerfData $HADOOP_JAVA_PLATFORM_OPTS"

# On secure datanodes, user to run the datanode as after dropping
privileges
export HADOOP_SECURE_DN_USER=${HADOOP_SECURE_DN_USER}

# Where log files are stored. $HADOOP_HOME/logs by default.
#export HADOOP_LOG_DIR=${HADOOP_LOG_DIR}/$USER

```

```
# Where log files are stored in the secure data environment.
export HADOOP_SECURE_DN_LOG_DIR=${HADOOP_LOG_DIR}/${HADOOP_HDFS_USER}

# The directory where pid files are stored. /tmp by default.
# NOTE: this should be set to a directory that can only be written to by
# the user that will run the hadoop daemons. Otherwise there is the
# potential for a symlink attack.
export HADOOP_PID_DIR=${HADOOP_PID_DIR}
export HADOOP_SECURE_DN_PID_DIR=${HADOOP_PID_DIR}

# A string representing this instance of hadoop. $USER by default.
export HADOOP_IDENT_STRING=$USER
```

### (5) roles/cdh5\_commons/templates/hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>dfs.replication</name>
    <value>{{ hadoop_hdfs['dfs_replication'] }}</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>{% for data_dir_parent in hadoop_hdfs['dfs_dir_parent'] %}
    {% if not loop.first and flag == 1 %},{% else %}{% set
    flag=1 %}{% endif %}file:///{{ data_dir_parent }}{{ hadoop_
    hdfs['dfs_dir_namenode'] }}{% endfor %}</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>{% for data_dir in hadoop_hdfs['dfs_dir_datanode'] %}{{
    if not loop.first and flag == 1 %},{% else %}{% set flag=1 %}
    {% endif %}file:///{{ data_dir }}{% endfor %}</value>
  </property>
  <property>
    <name>dfs.namenode.secondary.http-address</name>
    <value>{% for server in groups['secondarynamenode'] %}{% if not
    loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
    endif %}{{ server }}{% endfor %}:50090</value>
```

```

</property>
<property>
  <name>dfs.namenode.checkpoint.dir</name>
  <value>{% for data_dir in hadoop_hdfs['dfs_dir_parent'] %}{% if
    not loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
    endif %}file:///{{ data_dir }}{{ hadoop_hdfs['dfs_dir_sec_
    namenode'] }}{% endfor %}</value>
</property>
<property>
  <name>dfs.datanode.max.xcievers</name>
  <value>{{ hadoop_hdfs['dfs_datanode_max_xcievers'] }}</value>
</property>
</configuration>

```

#### (6) roles/cdh5\_commons/templates/mapred-site.xml

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

  <!--<property>
    <name>mapred.child.java.opts</name>
    <value>Xmx1024M</value>
  </property> -->

  <property>
    <name>mapreduce.map.memory.mb</name>
    <value>{{ hadoop_map_reduce['mr_map_mem_mb'] }}</value>
  </property>

  <property>
    <name>mapreduce.tasktracker.map.tasks.maximum</name>
    <value>{{ hadoop_map_reduce['mr_tt_map_task_max'] }}</value>
  </property>

  <property>

```



```

    <name>mapreduce.tasktracker.reduce.tasks.maximum</name>
    <value>{{ hadoop_map_reduce['mr_tt_reduce_task_max'] }}</value>
</property>

<property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>{{ hadoop_map_reduce['mr_reduce_mem_mb'] }}</value>
</property>

<property>
    <name>mapreduce.map.java.opts</name>
    <value>{{ hadoop_map_reduce['mr_map_java_opts'] }}</value>
</property>

<property>
    <name>mapreduce.reduce.java.opts</name>
    <value>{{ hadoop_map_reduce['mr_reduce_java_opts'] }}</value>
</property>

<property>
    <name>mapreduce.jobhistory.address</name>
    <value>{% for server in groups['jobhistoryserver'] %}{% if not
        loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
        endif %}{{ server }}{% endfor %}:10020</value>
</property>

<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>{% for server in groups['jobhistoryserver'] %}{% if not
        loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
        endif %}{{ server }}{% endfor %}:19888</value>
</property>
</configuration>

```

#### (7) roles/cdh5\_commons/templates/MasterScript.sh

```

# Update /etc/hosts

sudo cp /etc/hosts /etc/hosts.bkpz
sudo cp etc_new_hosts /etc/hosts

# Create Directories for NN/DN/JN

```

```
sudo mkdir -p /data1/nn /data1/jn
```

```
# Change Directory Permissions.
```

```
sudo chown hdadmin:hdadmin /data1/nn
```

```
sudo chown hdadmin:hdadmin /data1/jn
```

## (8) roles/cdh5\_commons/templates/slaves

```
{% for server in groups['datanodes'] %}
{{ server }}
{% endfor %}
```

## (9) roles/cdh5\_commons/templates/yarn-site.xml

```
<?xml version="1.0"?>

<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>

  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</
      name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>

  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>{% for server in groups['resourcemanager'] %}{% if not
      loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
      endif %}{{ server }}{% endfor %}</value>
  </property>

  <property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>{% for data_dir in hadoop_yarn['yarn_nodemgr_local_
      dir'] %}{% if not loop.first and flag == 1 %},{% else %}{%
      set flag=1 %}{% endif %}file:///{{ data_dir }}{% endfor %}</
      value>
```

```

</property>

<property>
  <name>yarn.nodemanager.log-dirs</name>
  <value>{% for data_dir in hadoop_yarn['yarn_nodemgr_log_dir'] %}{%
    if not loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
    endif %}file://{ {{ data_dir }}{% endfor %}</value>
</property>

<property>
  <name>yarn.log.aggregation-enable</name>
  <value>true</value>
</property>

<property>
  <description>Where to aggregate logs</description>
  <name>yarn.nodemanager.remote-app-log-dir</name>
  <value>hdfs://{% for server in groups['namenodes'] %}{% if not
    loop.first and flag == 1 %},{% else %}{% set flag=1 %}{%
    endif %}{{ server }}{% endfor %}:9000/var/log/hadoop_yarn/
    apps</value>
</property>

<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>{{ hadoop_yarn['yarn_nodemgr_resource_mem_mb'] }}</value>
</property>

<property>
  <name>yarn.scheduler.minimum-allocation-mb</name>
  <value>{{ hadoop_yarn['yarn_scheduler_min_alloc_mb'] }}</value>
</property>

<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>{{ hadoop_yarn['yarn_scheduler_max_alloc_mb'] }}</value>
</property>

<property>
  <name>yarn.log-aggregation-enable</name>
  <value>{{ hadoop_yarn['yarn_log_aggr_enable'] }}</value>
</property>

```

```

<property>
  <name>yarn.application.classpath</name>
  <value>
    {{ common['soft_link_base_path'] }}/hadoop/etc/hadoop/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      common/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      common/lib/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      hdfs/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      hdfs/lib/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      mapreduce/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      mapreduce/lib/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      yarn/*,
    {{ common['soft_link_base_path'] }}/hadoop/share/hadoop/
      yarn/lib/*
  </value>
</property>
</configuration>

```

## 2. 执行任务 task 的 playbook 脚本文件

创建 hadoop 用户账号，设置权限，该用户将用户对 Hadoop 环境的管理，包括服务的启动、重启、停止等。首先是创建操作系统的账号：

```

$cat ~/roles/cdh5_commons/tasks/main.yml
---
- name: Create a User '{{ hadoop_user }}' for all our Hadoop
  Modules.
  user: name={{ hadoop_user }} password={{ hadoop_password }}
- name: Create a .ssh Directory.
  file: path=~/.ssh state=directory owner={{ hadoop_user }} group=
    {{ hadoop_group }} mode=0700
  sudo: yes
  sudo_user: "{{ hadoop_user }}"
- name: Lets copy the template id_rsa to auth_keys location.
  template: src=id_rsa.pub dest=~/.ssh/authorized_keys mode=644

```

```

sudo: yes
sudo_user: "{{ hadoop_user }}"
- name: Lets copy id_rsa to location .ssh.
  template: src=id_rsa dest=~/.ssh/id_rsa mode=600
sudo: yes
sudo_user: "{{ hadoop_user }}"
- name: Lets copy id_rsa.pub to location .ssh.
  template: src=id_rsa.pub dest=~/.ssh/id_rsa.pub mode=644
sudo: yes
sudo_user: "{{ hadoop_user }}"

```

然后创建 Hadoop 的管理账号：

\$cat ~ /roles/cdh5\_commons/tasks/user.yml

```

---
- name: Create a User "{{ hadoop_user }}" for all our Hadoop
  Modules.
  user: name="{{ hadoop_user }}" password={{ hadoop_password }}

```

创建 Hadoop 安装目录，并把 Hadoop 的源文件解压，创建软链接如下所示：

\$cat ~ /roles/cdh5\_commons/tasks/install.yml

```

---
- name: Copy and UnArchive the Package in Destination Server.
  unarchive: creates={{ common['install_base_path'] }}/{{ hadoop_
    version }} src=file_archives/{{ hadoop_version }}.tar.gz
    dest={{ common['install_base_path'] }} owner={{ hadoop_user
    }} group={{ hadoop_group }}
- name: Change Directory Permissions.
  file: path={{ common['install_base_path'] }}/{{ hadoop_version }}
    owner={{ hadoop_user }} group={{ hadoop_group }} recurse=yes
- name: Creating a Symbolic Link in {{ common['install_base_path']
    }}/hadoop.
  file: src={{ common['install_base_path'] }}/{{ hadoop_version }}
    path={{ common['soft_link_base_path'] }}/hadoop state=link
    owner={{ hadoop_user }} group={{ hadoop_group }}

```

部署模板文件到每个节点上。

\$cat ~ /roles/cdh5\_commons/tasks/comfigure.yml



```

- name: Updating Configuration File in Zookeeper.
  template: src={{ item }} dest={{ common['soft_link_base_path']
    }}/hadoop/etc/hadoop/ owner={{ hadoop_user }} group={{
    hadoop_group }}
  with_items:
    - core-site.xml
    - hdfs-site.xml
    - yarn-site.xml
    - slaves
    - mapred-site.xml
    - hadoop-env.sh

```

这些模板就是 Hadoop 的配置文件，具体内容请参考《Hadoop 权威指南》。

### 11.3.2 部署 NameNode 角色

在 Hadoop 中，NameNode 负责对 HDFS 的元数据（metadata）持久化存储，处理来自客户端对 HDFS 各种操作的交互反馈。为了保证交互速度，HDFS 文件系统的元数据被转载到 NameNode 主机的内存中，并且会将内存中这些元数据保存到磁盘进行持久化存储。为了使这个持久化过程不会成为 HDFS 操作的瓶颈，Hadoop 通常不是对每一次操作的当前文件系统直接 snapshot 进行持久化，而是对 HDFS 最近一段时间的操作列表保存到 NameNode 中的 Editlog 文件中。当需要重启 NameNode 时，除了加载 fsImage 之外，还对 EditLog 文件中记录的 HDFS 操作进行重做（replay），恢复 HDFS 重启之前的最近状态。

部署 NameNode 包括主 NameNode 部署角色（cdh5\_namenode\_primary）和辅助 NameNode 部署角色（cdh5\_namenode\_secondary）。

#### 1. 部署主 NameNode

创建主 NameNode 的 hadoop\_hdfs 文件系统目录、辅助 NameNode 的 hadoop\_hdfs 文件系统目录，然后初始化 hadoop\_hdfs 文件系统，最后启动 Namenode 服务进程，代码如下所示：

```
$cat ~/roles/cdh5_namenode_primary/tasks/main.yml
```

```

---
- name: Create 'namenode' directory

```

```

file: path={{ item }}{{ hadoop_hdfs dfs_dir_namenode }} owner={{
    hadoop_user }} group={{ hadoop_group }} state=directory
with_items: hadoop_hdfs dfs_dir_parent

- name: Create 'secondary namenode' directory
  file: path={{ item }}{{ hadoop_hdfs dfs_dir_sec_namenode }} owner={{
    hadoop_user }} group={{ hadoop_group }} state=directory
  with_items: hadoop_hdfs dfs_dir_parent

- name: Format the namenode - [[ WILL NOT FORMAT IF current/VERSION]].
  command: creates={{ hadoop_hdfs dfs_dir_parent[0] }}{{ hadoop_hdfs.
    dfs_dir_namenode }}/current/VERSION sh {{ common['soft_link_
    base_path'] }}/hadoop/bin/hadoop namenode -format
  sudo: yes
  sudo_user: "{{ hadoop_user }}"

- name: Starting Namenode Service.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/hadoop-
    daemon.sh start namenode
  sudo: yes
  sudo_user: "{{ hadoop_user }}"

```

## 2. 部署辅助 NameNode

为提高 Hadoop 系统可靠性，生产系统一般还会部署辅助 NameNode，会周期性地 将 EditLog 中记录的 HDFS 操作合并到一个 CheckPoint 中，然后清空 EditLog。在 NameNode 重启时就会装载最新的一个 CheckPoint，并重做 EditLog 中记录的 HDFS 操作。由于 EditLog 中记录的是从上一次 CheckPoint 以后到现在的操作记录，所以比较小，能够快速恢复到重启 Hadoop 集群最近的状态，保证系统的完整性。

部署辅助 NameNode 角色与部署主 NameNode 的 playbook 脚本基本一样，只是在启动服务时候参数是 secondarynamenode，具体的 playbook 内容如下：

```

$cat ~/roles/cdh5_namenode_secondary/tasks/main.yml
---
- name: Create 'secondary-namenode' data directory.
  file: path={{ item }}{{ hadoop_hdfs dfs_dir_namenode }} owner={{
    hadoop_user }} group={{ hadoop_group }} state=directory
  with_items: hadoop_hdfs dfs_dir_parent

- name: Create 'secondary namenode' data directory

```

```
file: path={{ item }}{{ hadoop_hdfs.dfs_dir_sec_namenode }} owner={{
  hadoop_user }} group={{ hadoop_group }} state=directory
with_items: hadoop_hdfs.dfs_dir_parent
```

```
- name: Starting Secondary Namenode.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/hadoop-
    daemon.sh start secondarynamenode
  sudo: yes
  sudo_user: "{{ hadoop_user }}"
```

### 11.3.3 部署资源管理器角色

通过 YARN 大大扩展了 Hadoop 传统应用的潜在应用范围。YARN 构建于当前 Hadoop 集群的现有元素之上，是一个真正的 Hadoop 资源管理器，改进了 JobTracker 等元素，提高了可伸缩性和增强许多不同应用程序共享集群的能力，允许多个应用程序同时、高效地运行在一个的集群上。YARN 是大数据发展的一个基础性组件。YARN 将传统的 Hadoop 放到了一个可组合的、契合目的（fit-to-purpose）的平台中，以处理数据管理、分析和交易计算等工作。

YARN 中的资源管理器（Resource Manager）负责整个系统的资源管理和调度，并内部维护了各个应用程序的 ApplicationMaster 信息、NodeManager 信息、资源使用信息等。

在本例中，我们专门编写了启动资源管理器的角色，主要是启动 yarn 和 jobhistory 进程。详细脚本如下：

```
$cat ~ /roles/cdh5_resourcemgr/tasks/main.yml
```

```
---
- name: Start Resource Manager.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/
    yarn-daemon.sh start resourcemanager
  sudo: yes
  sudo_user: "{{ hadoop_user }}"

- name: Start Job History Server.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/mr-
    jobhistory-daemon.sh start historyserver
  sudo: yes
  sudo_user: "{{ hadoop_user }}"
```

### 11.3.4 部署 DataNode 角色

DataNode 是文件系统的工作节点，最终存储数据的位置。它们根据客户端或者 NameNode 的调度存储和检索数据，并且定期向 NameNode 发送它们所存储的块 (block) 的列表。

集群中的每个数据服务节点都运行着一个 DataNode 后台进程，这个后台进程负责把 HDFS 数据块读写到本地的文件系统。当需要通过客户端读 / 写某个数据时，先由 NameNode 告诉客户端去哪个 DataNode 进行具体的读 / 写操作，然后客户端直接与此 DataNode 服务节点的后台程序进行通信，并对相关的数据块进行读 / 写操作。

对于 DataNode 部署，我们编写了 `cdh5_datanode` 部署的角色，负责目录创建和启动后台进程。具体如下：

```
$cat ~ /roles/cdh5_datanode/tasks/main.yml
```

```
---
- name: Creating Datanode Directory.
  file: path={{ item }} owner={{ hadoop_user }} group={{ hadoop_group }} state=directory
  with_items: hadoop_hdfs dfs_dir_datanode

- name: Creating Yarn Local Directories.
  file: path={{ item }} owner={{ hadoop_user }} group={{ hadoop_group }} state=directory
  with_items: hadoop_yarn yarn_nodemgr_local_dir

- name: Creating Yarn Log Directories.
  file: path={{ item }} owner={{ hadoop_user }} group={{ hadoop_group }} state=directory
  with_items: hadoop_yarn yarn_nodemgr_log_dir

- name: Starting Datanode.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/hadoop-daemon.sh start datanode
  sudo: yes
  sudo_user: "{{ hadoop_user }}"

- name: Starting Node Manager.
  command: sh {{ common['soft_link_base_path'] }}/hadoop/sbin/yarn-daemon.sh start nodemanager
```

```
sudo: yes
sudo_user: "{{{ hadoop_user }}}"
```

在 DataNode 上启动 Hadoop 进程时将会调用从 `cdh5_common/templates` 部署到 `/opt/hadoop/etc/hadoop/yarn-site.xml` 的配置文件。

## 11.4 部署后 Hadoop 初始化与验证

Hadoop 系统部署完成后需要对 Hadoop 集群 DHFS 进行初始化, 同时运维也需要了解 Hadoop 运行状态, 可以从 Web 界面和命令行方式分别了解 Hadoop 集群的使用情况。

### 11.4.1 部署后初始化

安装好 Hadoop 软件、启动了服务之后, 就准备开始使用。这是需要对 Hadoop 集群系统做些初始化工作, 主要是创建 Hadoop 用户账号、授权、创建服务目录。专门放置在 `post_install_setups` 角色中, 具体内容如下:

```
$cat ~ /roles/post_install_setups/tasks/create_hadoop_user.yml
```

```
---
- name: Create a User '{{{ storm_user }}}' for all our Hadoop
  Modules.
  user: name="{{{ storm_user }}" password="{{{ storm_user }}"
- name: Create /tmp and /var directories in HDFS.
  command: sh "{{{ common['soft_link_base_path'] }}}/hadoop/bin/
    hadoop fs -mkdir -p {{{ item }}"
  sudo: yes
  sudo_user: "{{{ hadoop_user }}"
  with_items:
    - "/tmp"
    - "/var"
    - "/user/{{{ storm_user }}"
    - "/user/{{{ hadoop_user }}"
- name: Setting Permission for Hadoop /tmp and /var directories.
  command: sh "{{{ common['soft_link_base_path'] }}}/hadoop/bin/
    hadoop fs -chmod 1777 {{{ item }}"
  sudo: yes
  sudo_user: "{{{ hadoop_user }}"
  with_items:
```



```

- "/tmp"
- "/var"
- name: Setting Permission for Hadoop /user/{{ storm_user }}
  directory:
  command: sh "{{ common['soft_link_base_path'] }}" /hadoop/bin/
    hadoop fs -chown {{ storm_user }}:{{ storm_group }} /user/{{
      storm_user }}
  sudo: yes
  sudo_user: "{{ hadoop_user }}"

```

然后通过 main.yml 文件调用 create\_hadoop\_user.yml:

```
$cat ~ /roles/post_install_setups/tasks/main.yml
```

```

---
- include: create_hadoop_user.yml

```

## 11.4.2 部署后 Hadoop 验证

下面分别从 Web 界面和命令行方式了解 Hadoop 集群的使用情况。

### 1. Web 查看 Hadoop 集群信息

在浏览器中输入 <http://172.16.1.221:50070/>，将会看到 Hadoop 集群的主要信息，包括概览（OverView）、DataNodes、Datanode Volume Failures、Snapshot、Startup Progress、Utilities 等内容。图 11-2 是本例中集群概览的部分内容。

### 2. 命令行

创建 HDFS 文件目录：

```

$ /usr/local/hadoop-2.6.0-cdh5.4.7/bin/hadoop fs -mkdir /user/ansible
$ /usr/local/hadoop-2.6.0-cdh5.4.7/bin/hadoop fs -ls /user
Found 3 items
drwxr-xr-x - hdadmin      supergroup 0 2015-10-12 21:14 /user/ansible
drwxr-xr-x - hdadmin      supergroup 0 2015-10-11 22:39 /user/hdadmin
drwxr-xr-x - stormadmin  stormadmin 0 2015-10-11 22:39 /user/stormadmin

```

修改 HDFS 文件目录权限：

```

$ /usr/local/hadoop-2.6.0-cdh5.4.7/bin/hadoop fs -chown ansible:ansible
  /user/ansible
$ /usr/local/hadoop-2.6.0-cdh5.4.7/bin/hadoop fs -ls /user

```

Found 3 items

```
drwxr-xr-x - ansible ansible      0 2015-10-12 21:14 /user/ansible
drwxr-xr-x - hdadmin supergroup  0 2015-10-11 22:39 /user/hdadmin
drwxr-xr-x - stormadmin stormadmin 0 2015-10-11 22:39 /user/stormadmin
```

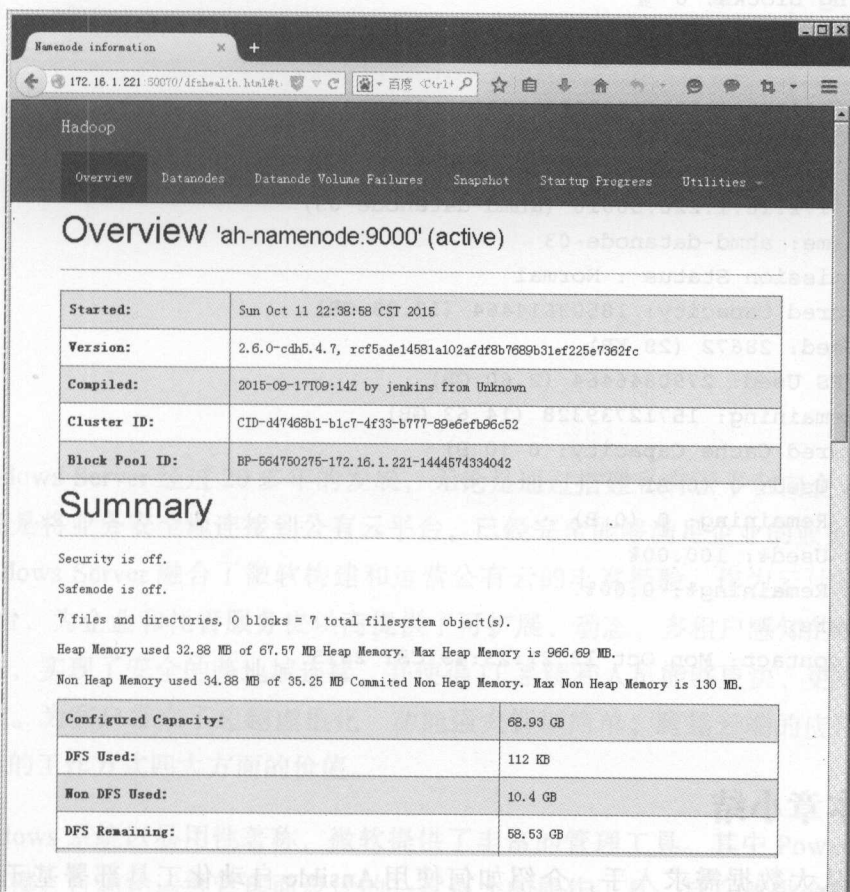


图 11-2 Hadoop 运行状态信息

显示输出 HDFS 文件系统信息：

```
$ /usr/local/hadoop-2.6.0-cdh5.4.7/bin/hadoop dfsadmin -report
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
```

Configured Capacity: 74014457856 (68.93 GB)

Present Capacity: 62851162112 (58.53 GB)

DFS Remaining: 62851047424 (58.53 GB)

```
DFS Used: 114688 (112 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
```

```
-----
Live datanodes (4):
```

```
Name: 172.16.1.226:50010 (ahmd-datanode-03)
Hostname: ahmd-datanode-03
Decommission Status : Normal
Configured Capacity: 18503614464 (17.23 GB)
DFS Used: 28672 (28 KB)
Non DFS Used: 2790846464 (2.60 GB)
DFS Remaining: 15712739328 (14.63 GB)
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Mon Oct 12 21:23:49 CST 2015
.....
```

## 11.5 本章小结

本章从大数据需求入手，介绍如何使用 Ansible 自动化工具部署基于 CDH5 的 Hadoop 环境。从操作系统安装、操作系统初始化、部署 Hadoop 软件，到最后验证部署系统的完整过程。按照每个功能一个角色的方式进行设计，展现一个完整的、复杂的系统如何进行自动化的工作，可以借鉴本章的内容快速在生产环境中使用。

## Ansible 管理 Windows 系统

Windows Server 经过 20 多年的发展，无论是通过搭建私有云平台向企业内部提供服务，还是将业务安全地连接到公有云平台，已经完全能够满足企业的业务与 IT 的挑战。Windows Server 融合了微软构建和运营公有云的丰富经验，作为云计算的新一代的 IT 平台，为企业和托管服务提供商提供了可扩展、动态、多租户感知的服务器和云基础架构，实现了安全的跨地域连接，并使得 IT 系统和人员能够更快、更有效地响应业务需求。为客户带来了超越虚拟化、功能强大管理简单、跨越云端的应用体验、现代化灵活的工作方式四大方面的价值。

Windows 系统以易用性著称，微软提供了丰富的管理工具。其中 PowerShell 是最适应大规模、自动化运维管理而设计的一款强大的操作工具，旨在改进命令行和脚本环境。PowerShell 以 .NET Framework 为平台，接收和返回 .NET 对象，此举为管理和配置微软系统带来了新的方法和工具。PowerShell 的版本随 Windows 的发布而更新，从最初 2006 年开始发布 1.0 版本之后陆续发布了 2.0、3.0、4.0，PowerShell 5.0 也即将随 Windows Server 2016 一起发布。

另外，微软从 Windows Server 2008 开始提供一个全新的 Server Core 模式，它是一个最小限度的系统安装选项，只包括安全、TCP/IP、文件系统、RPC 等服务器核心子系统。在 Server Core 仅有非常少的 GUI，可以安装 DNS、DHCP、文件服务、活动

目录、AD LDS（轻型目录服务）、打印、媒体、Web 这几种服务器角。Server Core 模式大大简化了维护管理、减少了安全攻击接触面、提高可用性、降低磁盘空间占用、较少的补丁安装，这是更加体现了 PowerShell 功能的强大和使用的便捷。

Ansible 从 1.7 版本开始支持对 Windows 管理，并将在 2.0 版本中支持的管理功能得到大幅度增强。本章将专门介绍如何使用 Ansible 及其模块，通过 PowerShell 来管理 Windows 系统，实现对 Windows 的自动化运维管理。主要介绍：

- 用 Ansible 管理 Windows 的基本原理
- 搭建 Ansible 管理 Windows 的基础环境
- 管理 Windows 的支持模块
- 常用 Windows 管理实例

## 12.1 Ansible 管理 Windows 工作原理

如前所述，Ansible 默认是通过 SSH 方式管理 Linux/UNIX 被管主机。但 Ansible 控制主机与被管主机之间的通信不再是 SSH 了，而是通过远程方式执行 Windows 内生 PowerShell 实现的。Ansible 管理远程主机还是继承管理 Linux/UNIX 系统时不需要代理端的特点，不需要在远程被管主机上安装额外软件。Ansible 的控制主机还是需要运行在 Linux 上，使用“WinRM”这个 Python 模块与远程 Windows 主机交互。

WinRM（即 Windows 远程管理）是 WS-Management 协议的 Microsoft 实现，该协议基于简单对象访问协议（SOAP）的、防火墙友好的标准协议，使来自不同供应商的硬件和操作系统能够互操作。WS-Management 协议由硬件和软件制造商群体开发，作为一种公共标准，可用于与实现该协议的任何计算机设备远程交换管理数据。

WinRM 旨在改进网络环境中的硬件管理，这种环境包含许多运行各种操作系统的设备。该服务的整体设计侧重于通过实现可互操作的标准协议来监视和管理远程计算机。WinRM 监听在 5985/5986 端口，该端口系统默认是关闭的，但是有些系统管理员为了便于他们对服务器进行远程管理，会将这个端口开启。具体的管理过程如图 12-1 所示。

当我们启用了 WinRM 时，你可以使用 PowerShell 对服务器进行远程查询和控制。当然，如果你觉得自己很有把握，也可以自行开启 WinRM。下面就是通过 PowerShell 指令来开启 WinRM 服务：



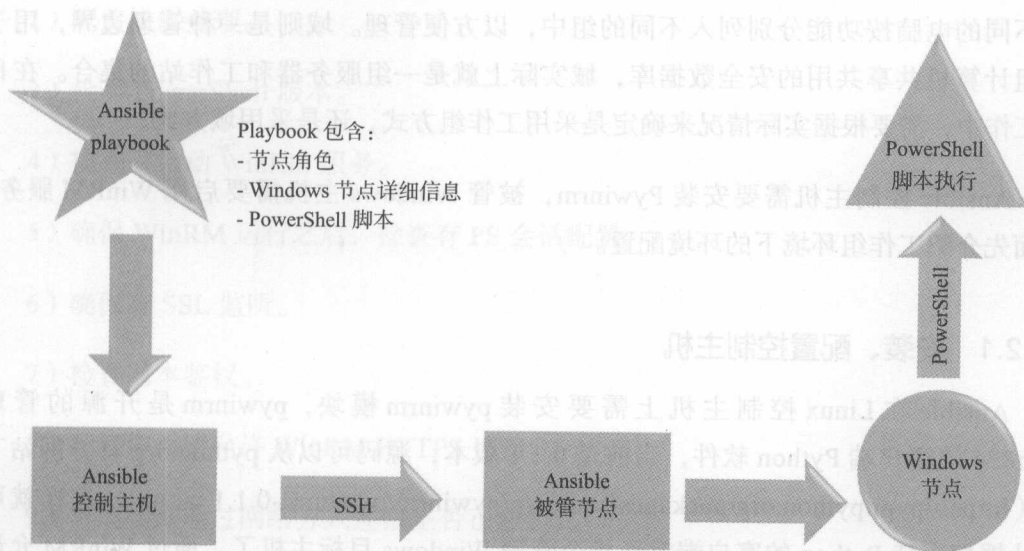


图 12-1 Ansible 管理 Windows 的工作原理

```
powershell Enable-PSRemoting -Force
```

然后你就可以看到输出结果如图 12-2 所示。

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Administrator> Enable-PSRemoting -Force
WinRM already is set up to receive requests on this machine.
WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.
WinRM Firewall exception enabled.
PS C:\Users\Administrator> _
```

图 12-2 输出结果

WinRM 要求操作人员的身份必须是受目标系统信任的。你可能需要一个域名使用者的令牌，这个令牌可以证明你是目标主机系统的本地管理员。你可以去偷一个令牌，或者使用 Runas 工具去生成一个令牌，也可以使用 Mimikatz 工具生成一个用于传递密码哈希值的令牌。

注意，学习 Ansible 管理 Windows 主机之前，最好能先了解前面介绍过的 playbook 相关知识。

## 12.2 搭建 Ansible 管理工作组 Windows 环境

对于 Windows 主机工作环境可分为工作组和域两种。工作组 (Work Group) 就是

将不同的电脑按功能分别列入不同的组中，以方便管理。域则是一种管理边界，用于一组计算机共享共用的安全数据库，域实际上就是一组服务器和工作站的集合。在日常工作中，需要根据实际情况来确定是采用工作组方式，还是采用域方式。

Ansible 控制主机需要安装 Pywinrm，被管 Windows 主机需要启用 WinRM 服务。下面先介绍工作组环境下的环境配置。

### 12.2.1 安装、配置控制主机

Ansible 在 Linux 控制主机上需要安装 pywinrm 模块，pywinrm 是开源的管理 WinRM 的客户端 Python 软件，当前是 0.1.1 版本，源码可以从 python.org 官方网站下载（<https://pypi.python.org/packages/source/p/pywinrm/pywinrm-0.1.1.tar.gz>），这样就可以从授权支持 Python 的客户端发送指令管理 Windows 目标主机了。通过 WinRM 允许调用 Windows 的本地对象，运行包括、不限于运行的批处理脚本、PowerShell 脚本、抓取 WMI 变量。

在 Ansible 控制主机上执行下面指令安装 pywinrm：

```
pip install "pywinrm>=0.1.1"
```

pywinrm 如果需要支持活动目录还需要安装 python-kerberos 模块，这将在专门讲解支持活动目录环境中讲解。

### 12.2.2 被管 Windows 主机配置

被管 Windows 主机需要满足两个条件：1）必须开启 WinRM 支持远程管理；2）确保 PowerShell 版本是 3.0 或更高版本。

#### 1. 开启 WinRM 支持远程管理

Ansible 为了方便用户配置 Windows 环境的 WinRM 服务，提供了一个初始化 Windows 环境的 PowerShell 脚本，可以从 Ansible 的 github 仓库中下载，地址是：<https://github.com/ansible/ansible/blob/devel/examples/scripts/ConfigureRemotingForAnsible.ps1>，然后在被管 Windows 主机上执行即可。这个脚本将完成如下操作：

- 1) 检查最后安装证书的指纹。

- 2) 配置错误处理。
- 3) 检测 PowerShell 版本。
- 4) 检查、启动 WinRM 服务。
- 5) 确保 WinRM 运行之后，检查有 PS 会话配置。
- 6) 确保有 SSL 监听。
- 7) 检查基本鉴权。
- 8) 配置防火墙允许 WinRM HTTPS 连接。
- 9) 本地测试通过网络方式连接是否正常。

这个脚本执行的输出结果如图 12-3 所示。

```
PS D:\work\github\ansible\examples\scripts> .\ConfigureRemotingForAnsible.ps1
在此计算机上设置了 WinRM 以接收请求。
WinRM 已经进行了更新，以用于远程管理。
在 HTTP:///* 上创建 WinRM 侦听器程序接受 WS-Man 对此机器上任意 IP 的请求。
WinRM 防火墙异常已启用。
已配置 LocalAccountTokenFilterPolicy 以远程向本地用户授予管理权限。

wxf      : http://schemas.xmlsoap.org/ws/2004/09/transfer
a        : http://schemas.xmlsoap.org/ws/2004/08/addressing
w        : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
lang     : zh-CN
Address  : http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous
ReferenceParameters : ReferenceParameters
确定。
```

图 12-3 脚本执行结果

注意，在执行这个脚本之前先运行 `Set $VerbosePreference = "Continue"`，这样可以看到执行的输出信息。

当然也可能期望稍微修改这个脚本，例如增加认证的时间信息。

**注意** 在 Windows7 和 Server 2008R2 主机上，由于在 Windows 管理 Framework 3.0 上有一个已知的 bug，可能需要安装补丁 <http://support.microsoft.com/kb/2842230> 来避免收到内存不足或堆栈溢出的异常。新安装的 Server 2008R2 系统如果没有升级到最新的补丁可能存在问题。

在 Windows 8.1 和 Server 2012 R2 上由于使用了 Windows 管理 Framework 4.0，因此没有受到这个问题的影响。

## 2. 确保 PowerShell 是 3.0 或更高版本

对于支持 Windows 的 Ansible 模块大多数需要 PowerShell 是 3.0 或以上软件版本，也是运行上述初始化脚本的必备环境。注意 PowerShell 3.0 只有在 Windows 7 SP1、Windows Server 2008 SP1、以及之后发行的 Windows 版本上支持。

查看 Ansible 查出的文件，拷贝 github.com 上 Ansible 官方脚本 `upgrade_to_ps3.ps1` 到远程的主机，以管理员的身份在 PowerShell 控制台上运行。下载脚本地址为 [https://github.com/cchurch/ansible/blob/devel/examples/scripts/upgrade\\_to\\_ps3.ps1](https://github.com/cchurch/ansible/blob/devel/examples/scripts/upgrade_to_ps3.ps1)。该脚本主要完成对当前被管 Windows 主机 PowerShell 版本的提取、判断，然后根据需要从微软的官网下载对应的 PowerShell 软件包，再进行升级。

当然，也可以直接从微软的官网上下载 Windows Management Framework 3.0 (<https://www.microsoft.com/en-us/download/details.aspx?id=34595>)，这个软件包包括 Windows PowerShell 3.0、WMI、WinRM、Management OData IIS Extension 和 Server Manager CIM Provider。

### 12.2.3 配置资源清单

支持 Windows 的 Ansible 需要有几个标准的变量来表示用户名、口令、远程主机的连接类型 (windows)。这些变量都很容易在资源清单中设置，这里使用的是口令方式而不是 SSH 密钥。简单在资源清单 `host` 文件中添加如下内容：

```
[windows]
win1.test.ansible.cn
win2.test.ansible.cn
```

在 Ansible 2.0 中，已经把 SSH 相关的变量 `ansible_ssh_user`、`ansible_ssh_host`、`ansible_ssh_port` 变成了 `ansible_user`、`ansible_host`、`ansible_port`。使用的 Ansible 版本是 2.0 之前，需要继续使用原来的变量风格 (`ansible_ssh_*`)，这些短的变量方式在 Ansible 旧版本中将被忽略，并且没有告警。

并在 `group_vars/windows.yml` 中定义了以下资源清单变量：

```
# it is suggested that these be encrypted with ansible-vault:
# ansible-vault edit group_vars/windows.yml
```



```

ansible_user: Administrator
ansible_password: SecretPasswordGoesHere
ansible_port: 5986
ansible_connection: winrm
# The following is necessary for Python 2.7.9+ when using default WinRM
self-signed certificates:
ansible_winrm_server_cert_validation: ignore

```

当使用 `playbook` 时候，记得要制定 `-ask-vault-pass` 来对带有口令的文件进行加密。

虽然 Ansible 通常是使用基于 SSH 的系统，但 Windows 管理一般不会使用 SSH。但 2015 年 6 月份一篇微软 PowerShell 团队发表的博客“Windows PowerShell Blog”(<http://blogs.msdn.com/b/powershell/archive/2015/06/03/looking-forward-microsoft-support-for-secure-shell-ssh.aspx>)中说明，PowerShell 团队意识到最好是把业界已经充分验证的 SSH 解决方案集成到 Windows 中，因此 PowerShell 团队将支持 OpenSSH 社区并做出贡献，提交 PowerShell 和 Windows SSH 解决方案。虽然这个项目还在规划阶段，也没有具体的交付时间，但 PowerShell 团队最近将提供详细的可用日期。

如果你已经安装了 Kerberos 模块、配置了 `ansible_user` (如 `username@realm`)，Ansible 首先将尝试用 Kerberos 认证。这种方式需要你的控制主机已经进行了 Kerberos 的认证，而不是通过“`ansible_user`”方式，如果这种方式失败，要么是你控制主机没有注册到 Kerberos 中，要么是远程主机上对应的域账号不可用。这时 Ansible 将会退回使用通常的用户名/口令方式进行认证。

自从版本 2.0 开始，下面这些定制的资源清单变量对于 WinRM 连接的配置也是支持的：

- `ansible_winrm_scheme`：指定使用 WinRM 连接时的连接方式，HTTP 或者 HTTPS，WinRM 的监听端口为 5985 (HTTP) 和 5986 (HTTPS)，Ansible 默认是使用 HTTPS。

- 1) 在 PowerShell 中输入 `winrm e winrm\config\listener` 来查看当前 WinRM 监听的端口，本例中显示监听 5985 和 5986 端口，如图 12-4 所示。

- 2) `set-item` 修改端口号，即 `set-item wsman:\localhost\listener\listener*\port 80`，把监听改为 80 端口。



```

PS C:\> winrm e winrm/config/listener
Listener
  Address = *
  Transport = HTTP
  Port = 5985
  Hostname
  Enabled = true
  URLPrefix = wsman
  CertificateThumbprint
  ListeningOn = 127.0.0.1, 192.168.1.9, ::1, fe80::100:7f:fffex13, fe80::5efe:192.168.1.9%12

Listener
  Address = *
  Transport = HTTPS
  Port = 5986
  Hostname = WIN01
  Enabled = true
  URLPrefix = wsman
  CertificateThumbprint = 674BDDFD676D2148EF6B3C4E1FAAE87B9B4E33F4
  ListeningOn = 127.0.0.1, 192.168.1.9, ::1, fe80::100:7f:fffex13, fe80::5efe:192.168.1.9%12

PS C:\>

```

图 12-4 在 PowerShell 中查看监听的端口

- `ansible_winrm_path`: 制定终端节点的 WinRM 路径, Ansible 默认使用 “/wsman”。
- `ansible_winrm_realm`: 指定实际使用的 Kerberos 认证方式, 如果用户名包含 “@”, Ansible 默认将会使用 @ 后面部分作为用户名。
- `ansible_winrm_transport`: 用逗号分隔一个或多个传输, 如果没有特别安装了 kerberos 模块、定义了 realm, 默认 Ansible 将用 Kerberos 的明文方式。
- `ansible_winrm_server_cert_validation`: 制定服务器认证方式: ignore 或 validate。在 Python 2.7.9 或更高版本中, Ansible 默认是 validate。除非你在 WinRM 监听器中特别配置, 否则采用 Windows 自签名方式的认证将会导致 Ansible 认证出错, 因此需要在配置文件中设置为 “ignore”。
- `ansible_winrm_*`: WinRM 协议支持的其他任何关键参数, 都可以支持。

## 12.2.4 测试被管 Windows 主机的连通性

经过前面对控制主机和被管节点的配置, 现在可以测试你的配置是否正确, 最简单的方式就是尝试连接你的 Windows 被管节点。注意这里不是 ICMP 的 ping, 但可以用 `win_ping` 模块来探测与远端 Windows 系统的 Ansible 通信通道:

```

#ansible windows -i host -m win_ping --ask-vault-pass
vault password:

```

```

win1.test.ansible.cn | success >> {
  "changed": false,
  "ping": "pong"
}
win2.test.ansible.cn | success >> {
  "changed": false,
  "ping": "pong"
}

```

如果出现 success 表示连通性良好，否则可以加上 -vvvv 参数查看详细错误信息，并参考下一节列出的常见问题处理方法。

## 12.2.5 常见问题处理

搭建 Ansible 管理 Windows 环境时候经常会出现 Windows 访问权限问题，下面介绍常见问题的处理方法。

### 1. 返回 401: Unauthorized. basic auth failed

现象：HTTP 返回错误 401.1，即未经授权，访问由于凭据无效被拒绝。

原因：由于用户匿名访问使用的账号（默认是 IUSR\_ 机器名）被禁用，或者没有权限访问计算机，将造成用户无法访问。

解决方案：

1) 查看 IIS 管理器中站点安全设置的匿名帐户是否被禁用，如果是，请尝试用以下办法启用：控制面板→管理工具→计算机管理→本地用户和组，将 IUSR\_ 机器名账号启用。如果还没有解决，请继续下一步。

2) 查看本地安全策略中，IIS 管理器中站点的默认匿名访问账号或者其所属的组是否有通过网络访问服务器的权限，如果没有尝试用以下步骤赋予权限：开始→程序→管理工具→本地安全策略→安全策略→本地策略→用户权限分配，双击“从网络访问此计算机”，添加 IIS 默认用户或者其所属的组。

注意：一般自定义 IIS 默认匿名访问账号都属于组，为了安全，没有特殊需要，请遵循此规则。

## 2. 返回 401.2

现象：HTTP 错误 401.2，即未经授权，被服务器配置拒绝访问。

原因：关闭了匿名身份验证

解决方案：

运行 `inetmgr`，打开站点属性→目录安全性→身份验证和访问控制→选中“启用匿名访问”，输入用户名，或者点击“浏览”选择合法的用户，并两次输入密码后确定。

## 3. 返回 401.3

现象：HTTP 错误 401.3，未经授权访问，ACL 对所请求资源设置了拒绝访问。

原因：IIS 匿名用户一般属于 `Guests` 组，而我们一般把存放网站的硬盘的权限只分配给 `administrators` 组，这时候按照继承原则，网站文件夹也只有 `administrators` 组的成员才能访问，导致 IIS 匿名用户访问该文件的 NTFS 权限不足，从而导致页面无法访问。

解决方案：

给 IIS 匿名用户访问网站文件夹的权限，即进入该文件夹的安全选项，添加 IIS 匿名用户，并赋予相应权限，一般是读、写。

## 12.3 搭建 Ansible 管理活动目录 Windows 环境

在一般企业中的 Windows 环境通常是采用域的方式部署，如果你希望使用目录服务管理的域账号连接（与远程本地创建的本地账号相对应），需要在 Ansible 控制主机上安装“`python-kerberos`”模块，当然还需要安装 MIT `krb5` 的依赖包。Ansible 被管主机也需要在目录服务中配置合适的账号。下面详细介绍如何部署支持管理 Windows 域的环境。

### 1. 安装 python-kerberos 依赖包

对于 Ansible 控制主机不同的运行环境 `python-kerberos` 依赖包的安装方式也略有差异，如下所示：

```

# Via Yum
yum -y install python-devel krb5-devel krb5-libs krb5-workstation

# Via Apt (Ubuntu)
sudo apt-get install python-dev libkrb5-dev

# Via Portage (Gentoo)
emerge -av app-crypt/mit-krb5
emerge -av dev-python/setuptools

# Via pkg (FreeBSD)
sudo pkg install security/krb5

# Via OpenCSW (Solaris)
pkgadd -d http://get.opencsw.org/now
/opt/csw/bin/pkgutil -U
/opt/csw/bin/pkgutil -y -i libkrb5_3

# Via Pacman (Arch Linux)
pacman -S krb5

```

## 2. 安装 python-kerberos 软件包

安装好必要的依赖包后，就可以通过 pip 安装 python-kerberos 软件包：

```
pip install kerberos
```

注：Kerberos 在 OS X 和许多发行的 Linux 系统上默认是已经安装了，如果你的控制主机还没有安装，需要自己安装。

## 3. 配置 Kerberos

编辑 /etc/krb5.conf，把下面需要连接的域相关信息添加上，这些配置是从 [realms] 开始。

```
[realms]
```

添加完整的域名、主目录服务 / 备用目录服务域控制器名称，如下：

```
[realms]
```

```
TEST.ANSIBLE.CN = {
```

```
    kdc = TEST.ANSIBLE.CN
```



```

    admin_server = TEST.ANSIBLE.CN
}

```

在 [domain\_realm] 的配置小节中添加你需要访问的每个域名信息，如下所示：

```

[domain_realm]
.test.ansible.cn = TEST.ANSIBLE.CN
test.ansible.cn = TEST.ANSIBLE.CN

```

在这里还可以配置如默认域信息等配置。

#### 4. 测试 Kerberos 连接

如果已经安装了 krb5-workstation (yum) 或 krb5-user (apt-get)，就可以用下面命令测试与域控服务器的授权连接是否正常：

```

[root@ansiblecontrol ~]# kinit wintest@TEST.ANSIBLE.CN
Password for wintest@TEST.ANSIBLE.CN:<输入该用户口令>

```

注意，域名部分必须是全称、且是大写字母。可以用 klist 命令查看你获取的凭证：

```

[root@ansiblecontrol ~]# klist
Ticket cache: KEYRING:persistent:0:0
Default principal: wintest@TEST.ANSIBLE.CN

Valid starting      Expires            Service principal
02/01/2016 23:29:55  02/02/2016 09:29:55  krbtgt/TEST.ANSIBLE.CN@TEST.
ANSIBLE.CN
    renew until 02/08/2016 23:29:47
[root@ansiblecontrol ~]#

```

#### 5. 解决 Kerberos 连接错误

如果不能连接 Kerberos，请按照下面方式检查：

- 确保你域名的 DNS 逆向解析是正常的。可以通过 ping 想要控制的 Windows 主机的主机名，然后在 nslookup 中查询该域名对应的 IP 地址，这两种情况需要返回一样的名称。如果返回的主机名与原来 ping 的方式返回的不同，与你负责活动目录的管理人员联系确保是支持 DNS 的逆向查询。
- 确保 Ansible 控制主机中有一个已经在域控制器中配置好的账号。
- 检查 Ansible 控制主机的时钟与你域控制服务器的时钟同步。Kerberos 对时间很



敏感,微小的时间差异可能产生 ticket 不被确认。

确保使用的是域中真实的完整域名,有时域名会设置成别名,可以用下面方式检查:

```
kinit -C wintest@TEST.ANSIBLE.CN
klist
```

如果用 klist 返回的域名与你请求的域名不一致,那你正在使用的是别名,这样就要修改 krb5.conf 配置文件,使用完整的域名而不是别名。

## 12.4 支持管理 Windows 模块

Ansible 从 1.7 版本(2014 年 8 月发行)开始支持对 Windows 的管理,支持的管理模块在经过 1.9 少量增加之后,今年 2016 年 1 月份发现的 2.0 版本中极大地增强了对 Windows 的管理功能,增加的管理模块超过之前所有模块的总和,当前官方发布的 Windows 模块总计已有 29 个,如表 12-1 所示。

表 12-1 官方发布的 Windows 模块数量

序号	Ansible 发行版本	增加 Windows 管理模块	发行时间
1	1.7	9	2014 年 8 月
2	1.9	4	2015 年 4 月
3	2.0	16	2016 年 1 月

这些 Windows 模块大致可以分为运行环境管理、运行服务管理、软件包管理、文件操作管理、IIS 配置管理、安全控制管理六大类,具体如表 12-2 所示。

表 12-2 官方发布的 Windows 模块类别

序号	类别	模块名称
1	运行环境管理	win_dotnet_ngen、win_environment(E)、win_regedit(E)、win_ping
2	运行服务管理	win_nssm(E)、win_service、win_scheduled_task(E)
3	软件包管理	win_chocolatey(E)、win_feature、win_msi、win_package(E)、win_webpicmd(E)
4	文件操作管理	win_copy(E)、win_file、win_get_url、win_lineinfile、win_template、win_unzip、win_stat
5	IIS 配置管理	win_iis_virtualdirectory(E)、win_iis_webapplication(E)、win_iis_webapppool(E)、win_iis_webbinding(E)、win_iis_website(E)
6	安全管理	win_acl(E)、win_firewall_rule(E)、win_group、win_user(E)、win_updates(E)

注:模块名称后面括号中的 E 表示该模块为扩展模块,没带标识的表示是核心模块。

表 12-3 介绍这些模块分别实现的功能以及 Ansible 软件开始支持的版本。

表 12-3 Windows 模块功能描述及 Ansible 支持的版本

模块名称	功能描述	支持版本	模块类型
win_acl	设置文件 / 目录权限	2.0	(E)
win_chocolatey	使用 chocolatey 安装软件包	1.9	(E)
win_copy	从一台 windows 主机拷贝文件到远程主机上	1.9.2	(E)
win_dotnet_ngen	在 .NET 升级之后用 ngen 重新编译 DLL 文件	2.0	
win_environment	修改 Windows 主机上的环境变量	2.0	(E)
win_feature	安装、卸载 Windows 特性功能。	1.7	
win_file	对文件或目录的操作，包括创建、修改、删除	1.9.2	
win_firewall_rule	对被管主机的 Windows 防火墙规则进行设置；	2.0	(E)
win_get_url	对给定的 URL 获取文件；	1.7	
win_group	增减或删除本地用户组；	1.7	
win_iis_virtualdirectory	配置 IIS 的虚拟目录	2.0	(E)
win_iis_webapplication	配置 IIS 的 Web 应用	2.0	(E)
win_iis_webapppool	配置 IIS 的 Web 应用池；	2.0	(E)
win_iis_webbinding	配置 IIS 的 Web 站点绑定信息	2.0	(E)
win_iis_website	配置 IIS 的 Web 站点	2.0	(E)
win_lineinfile	使用正则表达式规则检查文件中特定的行或对改行的替换	2.0	
win_msi	安装、卸载 Windows 的 MSI 安装软件	1.7	
win_nssm	用 NSSM 配置、管理应用服务	2.0	(E)
win_package	可以本地或从 url 下载可以安装的软件包进行安装，或卸载软件包；	1.7	(E)
win_ping	Windows 版本的 ping 模块	1.7	
win_regedit	添加、编辑、删除注册表注册项和设置值	2.0	(E)
win_scheduled_task	管理任务调度	2.0	(E)
win_service	管理 Windows 服务	1.7	
win_stat	获取 Windows 文件的详细信息	1.7	
win_template	把模版文件发送到远程服务器	1.9.2	
win_unzip	对 Windows 节点上的文件进行压缩或解压	2.0	(E)
win_updates	下载、安装 Windows 更新	2.0	(E)
win_user	管理本地的 Windows 账号	1.7	
win_webpicmd	使用 Web 平台命令行安装软件包	2.0	(E)

12.5 常用 Windows 管理实例

Ansible 对 Windows 管理的基本操作，可以用 Ad-hoc 方式命令行直接调用 Ansible 模块，也可以通过 playbook 方式进行操作。下面介绍常见的 Ansible 管理 Windows 模

块的基本应用，主要包括防火墙配置、Windows 角色及特性安装、配置 IIS 站点、网络文件下载及部署、Windows 服务管理等实例，更多的内容可以参考 Ansible 官网的文档。

## 1. 配置 Windows 防火墙

在 Windows 防火墙上打开 smtp 服务的 25 端口：

```
- name: Firewall rule to allow smtp on TCP port 25
  action: win_firewall_rule
  args:
    name: smtp
    enabled: yes
    state: present
    localport: 25
    action: allow
    protocol: TCP
```

## 2. 安装 Windows 角色和特性

通过 Ansible 安装 IIS 服务。同 Ad-hoc 命令行安装方式如下：

```
$ ansible -i hosts -m win_feature -a "name=Web-Server" all
$ ansible -i hosts -m win_feature -a "name=Web-Server,Web-Common-Http"
all
```

通过 playbook 方式如下：

```
- name: Install IIS
  hosts: all
  gather_facts: false
  tasks:
    - name: Install IIS
      win_feature:
        name: "Web-Server"
        state: present
        restart: yes
        include_sub_features: yes
        include_management_tools: yes
```

当然，还可以通过远程执行 PowerShell 脚本实现：

```
# PS C:\Users\Administrator> Import-Module ServerManager; Get-
WindowsFeature
```

### 3. 配置 IIS Web 站点

添加、删除、配置 IIS Web 站点。

获取已有站点信息如下所示：

```
# This return information about an existing host
$ ansible -i vagrant-inventory -m win_iis_website -a "name='Default Web
  Site'" window
host | success >> {
  "changed": false,
  "site": {
    "ApplicationPool": "DefaultAppPool",
    "Bindings": [
      " *:80:"
    ],
    "ID": 1,
    "Name": "Default Web Site",
    "PhysicalPath": "%SystemDrive%\inetpub\wwwroot",
    "State": "Stopped"
  }
}
```

停止已有站点服务如下所示：

```
# This stops an existing site.
$ ansible -i hosts -m win_iis_website -a "name='Default Web Site'
  state=stopped" host
```

添加新的 Web 站点如下所示：

```
# This creates a new site.
$ ansible -i hosts -m win_iis_website -a "name=acme physical_path=c:\
  sites\acme" host
```

指定 Web 站点日志的存放位置如下所示：

```
# Change logfile .
$ ansible -i hosts -m win_iis_website -a "name=acme physical_path=c:\
  sites\acme" host
```

用 playbook 方式配置 IIS 站点如下所示：

```
# Playbook example
```

```
---
- name: Acme IIS site
  win_iis_website:
    name: "Acme"
    state: started
    port: 80
    ip: 127.0.0.1
    hostname: acme.local
    application_pool: "acme"
    physical_path: 'c:\sites\acme'
    parameters: 'logfile.directory:c:\sites\logs'
  register: website
```

#### 4. 从网络下载部署文件

不同的脚本方式下载一个图片，保存在 C:\Users\RandomUser\ 目录中。

Ad-hoc 命令方式如下所示：

```
$ ansible -i hosts -c winrm -m win_get_url -a "url=http://www.example.
com/ansible.jpg dest='C:\Users\Administrator\ansible.jpg'" all
```

Playbook 方式下载如下所示：

```
# Playbook example
- name: Download ansible.jpg to 'C:\Users\RandomUser\ansible.jpg'
  win_get_url:
    url: 'http://www.example.com/ansible.jpg'
    dest: 'C:\Users\RandomUser\ansible.jpg'
```

Playbook 方式发生变化时才下载：

```
- name: Download ansible.jpg to 'C:\Users\RandomUser\earthrise.jpg' only
  if modified
  win_get_url:
    url: 'http://www.ansible.cn/ansible.jpg'
    dest: 'C:\Users\RandomUser\ansible.jpg'
    force: no
```

代理方式下载：

```
- name: Download ansible.jpg to 'C:\Users\RandomUser\ ansible.jpg'
```



```
through a proxy.
```

```
win_get_url:
```

```
  url: 'http://www.ansible.cn/ansible.jpg'
```

```
  dest: 'C:\Users\RandomUser\ansible.jpg'
```

```
  proxy_url: 'http://10.0.0.1:8080'
```

```
  proxy_username: 'username'
```

```
  proxy_password: 'password'
```

通过 playbook 方式下载一个简单的 HTML，然后部署到 IIS 上，如下所示：

```
- name: Download simple web site
```

```
  hosts: windows
```

```
  gather_facts: false
```

```
  tasks:
```

```
    - name: Download simple web site to 'C:\inetpub\wwwroot\ansible.
      html'
```

```
      win_get_url:
```

```
        url: 'https://raw.githubusercontent.com/mywebapp/
          master/index.html'
```

```
        dest: 'C:\inetpub\wwwroot\ansible.html'
```

## 5. 管理 Windows 服务，包括启动、停止

重启 spooler 服务如下所示：

```
# Restart a service
```

```
win_service:
```

```
  name: spooler
```

```
  state: restarted
```

设置 spooler 服务开机自启动如下所示：

```
# Set service startup mode to auto and ensure it is started
```

```
win_service:
```

```
  name: spooler
```

```
  start_mode: auto
```

```
  state: started
```

## 12.6 本章小结

本章首先介绍了 Ansible 通过 WinRM 方式对 Windows 主机的管理。然后分别讲解对 Windows 工作组和域两种管理方式下 Ansible 如何管理，如何部署 Ansible 控制主机

和被管主机。之后又归纳、总结介绍了 Ansible 对 Windows 管理的支持模块。最后介绍 Ansible 如何支持这些 Windows 常用的管理操作。

Ansible 对 Windows 从 1.7 开始支持, Ansible 被 Redhat 收购之后, 重点增强了对 Windows 的支持, 自从 2.0 版本之后已经较为全面支持对 Windows 系统的管理, 随着 Ansible 版本的发展, 将会更多地支持管理 Windows 的模块。同时 Windows 系统本身也将对 SSH 的管理方式提供支持, 将来会更完美地融合 Ansible 对 Windows 的管理。这样, Ansible 将成为真正的跨平台自动化运维管理工具。

## 网络自动化管理的应用实战

运维自动化在企业 IT 部门中是高速增值的技术，能够同时自动化部署、创建、扩展应用和服务，很容易达到几百台、甚至上万台的水平，DevOps 和 SysOps 显示出了强大的功力。但基础架构中的网络又该如何管理呢，如何做好 NetOps 呢？

本章将带你一起来进入管理大量网络设备的 NetOps。首先回顾网络运维管理方式的发展历程，然后介绍 Ansible 及其集成的网络管理模块，最后介绍一些常用的第三方网络配置管理模块。

### 13.1 网络管理也自动化了

网络的配置从 1990 到 2010 近 20 年时间内没有发生太大的变化，只是功能更多时，有更多的配置命令。但是在最近的五年中，数据中心的网络发生了翻天覆地的变化：核心连接速度从千兆向 10G、40G 发展，连接速度的高速发展超过了交换机的连接速度和端口密度。但数据中心网络交换机的配置却还是原来的传统方式，并没有因设备的升级而变化，如图 13-1 所示。

命令行界面仍是核心网络设备配置的重要工具。一些网络设备供应商也有开始提供基于 Web 的图形界面配置和管理设备，试图可以从单一的客户端来处理整个网络设置，但这样并没有简化很多配置工作，只是把命令行换成 GUI 界面而已。

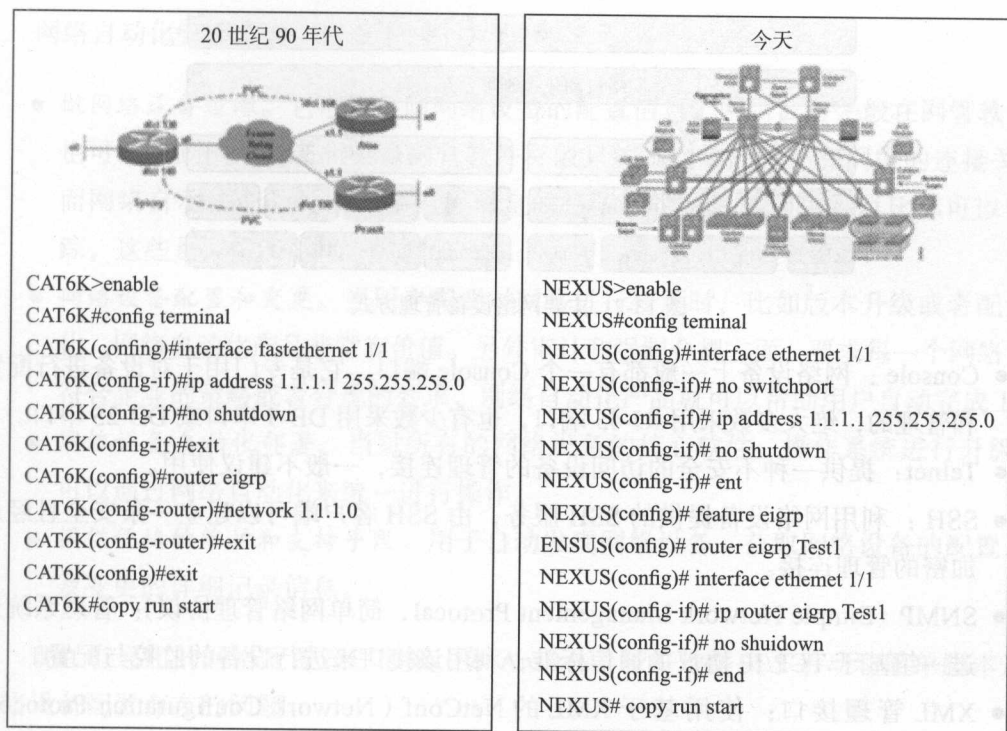


图 13-1 数据中心网络交换机的配置方式对比

许多企业的 IT 人员还是用手动配置的方式管理数以千计的端口。这看起来似乎不是什么大问题。当网络工程师配置一个网络中的设备的时候，他们还必须配置相应的网络接口。在大多数情况下，网络工程师配置好网络之后就用于网络中并没有什么问题，但是在虚拟化的现代化系统环境中，一台刀片服务器看似只有少数几个网络接口，却承载着数以百计的虚拟机。

然而问题并不是接入端口配置那么简单。你试想当 NTP（网络时间协议）服务器或认证服务器更改之后引发的问题吗？在大多数情况下，网络管理员手动登录到每个设备和配置的基础上设置这些服务器配置。一些熟练的网络专家虽然可以用脚本完成这些工作，但也会产生一些新的问题。因为无论通过哪种方法更改 NTP 服务器，这些改变是要覆盖全平台的。

当前主要的网络设备管理方式有 Console、Telnet、SNMP、XML 等管理方式，如图 13-2 所示。



图 13-2 主要网络设备管理方式

- **Console**：网络设备上一般都有一个 Console 端口，它是专门用于对设备进行配置和管理的，绝大多数采用 RJ-45 端口，也有少数采用 DB-9 串口或 DB-25 串口。
- **Telnet**：提供一种不安全的访问设备的管理连接，一般不建议使用。
- **SSH**：利用网络设备提供的 SSH 服务，由 SSH 客户端可以建立一条安全且经过加密的管理连接。
- **SNMP**（Simple Network Management Protocol，简单网络管理协议）：管理系统通过一组基于 TCP/IP 协议的通信标准，利用该接口来进行设备的监控与配置。
- **XML 管理接口**：使用基于 XML 的 NetConf（Network Configuration Protocol，网络配置协议），NetConf 基于 RFC 4741 的标准规范，利用 XML 的管理工具或管理程序，通过该接口来实现设备的管理、监控和通信。

当前业界的网络管理协议主要是 SNMP 和 NetConf。SNMP 采用 UDP，实现简单，技术成熟，但是在安全可靠、管理操作效率、交互操作和复杂操作实现上还不能满足管理需求。NetConf 采用 XML 作为配置数据和协议消息内容的数据编码方式，采用基于 TCP 的 SSHv2 进行传送，以 RPC 方式实现操作和控制。XML 可以表达复杂、具有内在逻辑、模型化的管理对象，如端口、协议、业务以及之间的关系等，提高了操作效率和对象标准化；采用 SSHv2 传送方式，可靠性、安全性、交互性较好。两种协议的对比如表 13-1 所示。

表 13-1 SNMP 和 NetConf 协议对比

关注点	SNMP	NetConf/XML
客户端可编程能力	No	灵活编程，但解释复杂
跨平台管理能力	No	Yes
批量配置	No	Yes，基于 XML 树组织管理请求
多样化操作手段	Set	Create/Merge/Delete/Replace/None, 简化客户端操作难度
过滤监控数据能力	No	Yes, 只看需要看的数据



网络自动化管理主要体现以下四个方面：

- 做网络设备追踪。它可以存储网络设备的配置信息，网络设备一般在网管软件里也可以看到网络设备，但是网管软件一般只关心网络的拓扑和网络的连接关系，而网络自动化产品可以存储配置信息的快照，对配置信息进行对比都可以做追踪，这些是网管产品所不包含的功能。
- 网络设备配置和变更。当用户需要对网络进行变更时，比如版本升级或者配置变化，网络自动化产品非常有价值。另外审计和强制合规方面，要求每一个网络设备符合企业的策略或者合规的要求，网络自动化产品就可以帮助用户自动完成工作。
- 网络设备自动化部署。当对所有的网络设备的核心软件、操作系统进行升级时，可以通过网络自动化来统一进行操作。
- 提供便捷的维护和支持手段。用于自动发现网络设备、获取网络设备的配置，以及变更的详细记录信息。

现在有通用的网络配置管理框架，如 Ansible，较好地解决了手动和简单脚本进行网络设备配置存在的问题。

## 13.2 Ansible 官方集成的网络角色

Ansible 已经包含了网络配置模块，官方网站文档 ([docs.ansible.com](https://docs.ansible.com)) 有详细介绍。由于网络自动化处于相对初级阶段，各设备厂家的自动化手段有一定的差异，当前 Ansible 官方支持的模块主要是以“extra”模块提供。“extra”模块是与 Ansible 一起发行，但模块相对较新，可能比对应的“core”模块维护、更新的活跃度低。

当前官方支持的网络模块主要包括：网络基础支撑模块和负载均衡模块。

网络基础支撑模块有：

- dnssimple：调用 dnssimple.com (DNS 主机服务) 接口查询服务。
- dnsmadeeasy：调用 dnsmadeeasy.com (DNS 主机服务) 接口查询服务。
- haproxy：配置 HAProxy 后端的服务，包括启用、失效、权重值设置等。
- ipify\_facts：查询公网 IP 的互联网网关。
- lldp：获取详细报表。

- nmcli : NetworkManager 用于管理包括 Ethernet、VLANS、Bridges、Bonds、Teams、Wi-Fi、Mobile Broadband (如移动 3G) 以及 IP-over-InfiniBand 等类型的连接, 可以配置网络别名、IP 地址、静态路由、DNS、VPN 连接等很多特殊参数。
- get\_url: 从 HTTP、HTTPS、FTP 下载文件到主机。
- slurp: 从远端节点提取一个文件。
- uri: 与 Web 服务进行交互。
- snmp\_facts: 使用 SNMP 查询设备的 facts。
- openvswitch\_bridge: 管理 Open vSwitch 的桥接配置。
- openvswitch\_db: 配置 open vswitch 的数据库。
- openvswitch\_port: 配置 Open vSwitch 端口。

负载均衡模块包括支持 A10、F5、Citrix 等:

- a10\_server: 管理 A10 的 AX/SoftAX/Thunder/vThunder 设备。
- a10\_service\_group: 管理 A10 网络设备的服务组。
- a10\_virtual\_server: 管理 A10 网络设备的虚拟服务器。
- netScaler: 管理 Citrix 的 NetScaler 实体配置。
- bigip\_facts: 收集 F5 BIG-IP 设备的 facts。
- bigip\_gtm\_wide\_ip: 管理 F5 BIG-IP GTM 的 wide ip。
- bigip\_monitor\_http: 管理 F5 BIG-IP LTM 的 http 监控。
- bigip\_monitor\_tcp: 管理 F5 BIG-IP LTM 的 tcp 端口监控。
- bigip\_node: 管理 F5 BIG-IP LTM 节点配置。
- bigip\_pool: 管理 F5 BIG-IP LTM 的 pool 配置。
- bigip\_pool\_member: 管理 F5 BIG-IP LTM 的 pool 成员。

下面是 F5 BIG-IP LTM 的 TCP 端口监控的 playbook:

```
- name: BIGIP F5 | Create TCP Monitor
  local_action:
    module: bigip_monitor_tcp
    state: present
    server: "{{ f5server }}"
    user: "{{ f5user }}"
    password: "{{ f5password }}"
```

```

name: "{{ item.monitorname }}"
type: tcp
send: "{{ item.send }}"
receive: "{{ item.receive }}"
with_items: f5monitors-tcp
- name: BIGIP F5 | Create TCP half open Monitor
  local_action:
    module: bigip_monitor_tcp
    state: present
    server: "{{ f5server }}"
    user: "{{ f5user }}"
    password: "{{ f5password }}"
    name: "{{ item.monitorname }}"
    type: tcp
    send: "{{ item.send }}"
    receive: "{{ item.receive }}"
  with_items: f5monitors-halftcp
- name: BIGIP F5 | Remove TCP Monitor
  local_action:
    module: bigip_monitor_tcp
    state: absent
    server: "{{ f5server }}"
    user: "{{ f5user }}"
    password: "{{ f5password }}"
    name: "{{ monitorname }}"
  with_flattened:
- f5monitors-tcp
- f5monitors-halftcp

```

更多关于 Ansible 官方支持的网络模块详见官网各个模块的介绍和使用方式。

### 13.3 生成配置文件及部署

你可以直接使用 Ansible 的 raw 模块执行网络设备上的命令。真正能够发挥 Ansible 强大功能的是使用基于 Jinja2 的模板系统来创建网络设备的配置文件。本节将介绍使用 Ansible 的模板、变量功能生成网络设备配置文件，然后再通过 SCP 加载到网络设备，这样能够对网络设备的配置信息进行集中管理、巡检配置变化。

### 13.3.1 生成网络配置模板

我曾经在生产环境中遇到需要同时配置 20 台 H3C VSR 做 L2TP VPN 拨号服务，接受来自超过 20 万并发的拨号请求。我们在规划阶段已经按照不同地区的客户端拨号到特定的 VSR 上，这样每台 VSR 的差异主要在拨号的 IP 地址、分配的地址池等细节上。在这里将简单介绍如何生成 VSR 的配置文件，然后再通过下一节的 SCP 方式加载配置文件。

我的 /etc/Ansible/hosts 文件只是配置了使用 localhost 方式：

```
[gituser@ip ~]$ cat /etc/ansible/hosts
#[local]
localhost ansible_ssh_user=admin
```

admin 用户既是本地运行 playbook 的账号，也是用于“远程”SSH 到 VSR 的用户。这里所谓的“远程”只是 SSH 到 localhost。

在用户 admin 的 /home/admin/ 创建 ANSIBLE/VSR-TEMPLATE 和 ANSIBLE/CFGS 两个目录，在 VSR-TEMPLATE 创建产生配置文件的 playbook 脚本，把产生的配置文件保存在 CFGS 下。在 VSR-TEMPLATE 按照 Ansible 实践创建下面文件：

- ./site.yml：产生结果的 playbook。
- ./roles/vsr/vars/main.yml：配置模板中设置的变量参数。
- ./roles/vsr/templates/vsr.j2：配置文件模板。
- ./roles/vsr/tasks/main.yml：角色执行的入口。

文件 ./site.yml 内容如下，就是一般的 playbook，调用了 vsr 角色：

```
---
- name: Generate VSR configuration files
  hosts: localhost
  roles:
    - vsr
```

角色 vsr 执行的 ./roles/vsr/tasks/main.yml 文件，将按照 test\_vsrs 中保存的变量，调用 vsr.j2 模板产生配置文件：

```
---
```

```
- name: Generate configuration files
  template: src=vsr.j2 dest=/home/admin/ANSIBLE/CFGs/{{item.hostname}}.txt
  with_items: test_vsr
```

roles\_test/vsr/vars/main.yml 保存 test\_vsr 中具体的变量:

```
---
test_vsr:
- { hostname: VSR01,ip_pool_start: 10.10.0.2,item.ip_pool_end:
  10.10.63.254,ip_address_virtual_template1: 10.10.0.1, ip_
  netmask_virtual_template1: 255.255.224.0 , ip_address_l2tp:
  192.168.251.11, ip_netmask_l2tp: 255.255.255.0, route_dest_
  ip: 0.0.0.0, route_hop_ip: 192.168.251.1,l2tp_admin_username:
  admin, l2tp_admin_user_passwd_hash: $6$7Rf0Q1toxUd9qWt7$Zpxx3
  ALbZdE8ZkVIOhGHxU9QWJZQ0ghGW6yJHXtBjo7ZwRapJHwbmsAvHsTnwg==,
  l2tp_admin_user_operator_level: level-15, l2tp_admin_user_role:
  network-operator, l2tp_username: fujian, l2tp_user_passwd_
  cipher: $c$3$Ru+Irtk0rlyYop6JePuFWgRw6b3nw== , l2tp_user_role:
  network-operator }
- { hostname: VSR02,ip_pool_start: 10.10.64.2,item.ip_pool_end:
  10.10.127.254,ip_address_virtual_template1: 10.10.64.1, ip_netmask_
  virtual_template1: 255.255.224.0 , ip_address_l2tp: 192.168.251.12,
  ip_netmask_l2tp: 255.255.255.0, route_dest_ip: 0.0.0.0, route_hop_
  ip: 192.168.251.1,l2tp_admin_username: admin, l2tp_admin_user_
  passwd_hash: $6$7Rf0Q1toxUd9qWt7$Zpxx3ALbZdE8ZkVIOhGHxU9QWJZQ0ghG
  W6yJHXtBjo7ZwRapJHwbmsAvHsTnwg==, l2tp_admin_user_operator_level:
  level-15, l2tp_admin_user_role: network-operator, l2tp_username:
  guangdong, l2tp_user_passwd_cipher: $c$3$Ru+Irtk0rlyYop6JePuFWgRw6
  b3nw== , l2tp_user_role: network-operator }
```

H3C VSR 的配置模板文件 ./roles\_test/vsr/templates/vsr.j2 如下所示:

```
#
version 7.1.059, ESS 0321
sysname {{ item.hostname }}
ip pool 1 {{ item.ip_pool_start }} {{ item.ip_pool_end }}
password-recovery enable
vlan 1
irf-port
interface Virtual-Temlate1
  ppp authentication-mode chap domain system
```



```

ppp timer negotiate 10
remote address pool 1
ip address {{ item.ip_address_virtual_template1 }} {{ item.ip_netmask_
    virtual_template1 }}
tcp mss 1400
interface NULL0
interface GigabitEthernet1/0
port link-mode route
ip address {{ item.ip_address_l2tp }} {{ item.ip_netmask_l2tp }}
interface GigabitEthernet2/0
port link-mode route
scheduler logfile size 16
line class aux
user-role network-operator
line class console
user-role network-admin
line class vty
user-role network-operator
line aux 0
user-role network-operator
line con 0
user-role network-admin
line vty 0 63
authentication-mode scheme
user-role network-operator
ip route-static {{ item.route_dest_ip }} {{ item.route_hop_ip }}
ssh server enable
domain system
authentication ppp local
aaa session-limit ssh 32
domain default enable system
role name level-0
description Predefined level-0 role
.....< 省略 level-1 ~ level-13 内容 >.....
role name level-14
description Predefined level-14 role
user-group system
local-user {{ item.l2tp_admin_username }} class manage
password hash {{ item.l2tp_admin_user_passwd_hash }}
service-type ssh
authorization-attribute user-role {{ item.l2tp_admin_user_operator_level }}
authorization-attribute user-role {{ item.l2tp_admin_user_role }}

```

```

local-user {{ item.l2tp_username }} class network
password cipher {{ item.l2tp_user_passwd_cipher }}
service-type ppp
authorization-attribute user-role {{ item.l2tp_user_role }}
l2tp enable
return

```

在执行 /home/admin/ANSIBLE/VSR-TEMPLATE 目录下, 执行 \$ansible-playbook site.yml, 将在 /home/admin/ANSIBLE/CFGs 目录下获得 VSR01.txt~VSR20.txt 对应的配置文件。

### 13.3.2 部署配置模板

如何把前一节生产的配置文件加载到网络设备上, 可以通过利用 Netmiko 组件, 从 <https://github.com/ktbyers/netmiko> 下载。Netmiko 是基于 Paramiko 的 SSH 连接类库, 大大简化了 SSH 管理网络设备, 也因此得名。主要完成以下工作:

- 建立与网络设备的 SSH 连接。
- 简化执行 show 命令以及返回输出数据。
- 简化执行配置命令, 包括 commit 动作。
- 支持不同厂商、不同平台的网络设备。

到当前, Netmiko 已经支持主流网络设备平台, 包括:

- Cisco: IOS、IOS-XE、ASA、NX-OS、Cisco IOS-XR、Cisco WLC (limited testing)
- Arista: EOS
- HP: ProCurve、Comware (limited testing)
- Juniper: Junos
- Brocade: VDX (limited testing)
- F5: LTM (experimental)
- 华为: Huawei (limited testing)

在使用 cisco\_file\_transfer 的 Ansible 模块时, 管理节点需要安装 Ansible1.8.2、Python 2.7.5、Paramiko 1.14.0、Python scp module 0.10.0、Netmiko 0.2.0 等软件。

定义 Ansible 资源清单文件如下:

```
[local]
localhost ansible_connection=local

[cisco]
pynet-sw1 host=192.168.1.100 port=22 username=admin password=password

[cisco:vars]
ansible_python_interpreter=~/.applied_python/bin/python
ansible_connection=local
```

通过执行 `./load_file.yml` 文件，由 SCP 把配置文件 `cisco_logging.txt` 加载到 Cisco Catalyst 2960 的 flash 上：

```
---
- name: Cisco IOS testing
  hosts: cisco
  gather_facts: False

  tasks:
    - name: Testing file transfer
      cisco_file_transfer:
        source_file="/home/admin/scp_sidecar/tests/cisco_logging.txt"
        dest_file=cisco_logging.txt
        enable_csp=true
        host={{ host }}
        port={{ port }}
        username={{ username }}
        password={{ password }}
```

这个 playbook 脚本执行过程如下：

```
[admin@test]$ansible-playbook -v load_file.yml
PLAY [Cisco IOS testing]*****
TASK: [Testing file transfer]*****
Changed: [cisco01]=>{"changed":true,"msg": "File successfully
transferred to remote device"}
PLAY RECAP *****
Pynet-rtr1 : ok=1 changed=1 unreachable=0 failed=0
```

到 Cisco 交换机上验证下加载结果：

```
Cisco01#dir flash:cisco_logging.txt
```

```
Directory of flash:/cisco_logging.txt
15 -rw- 420 Oct 29 2015 13:14:36 -07:00 cisco_logging.txt
```

## 13.4 通过 SNMP 方式配置网络

我们可能已经对 Cisco IOS 设备很熟悉，并且前几年大量部署在生产环境中，但是这些 Cisco IOS 设备缺少现代的 API。如果你想使用 Ansible，需要有 SSH、SNMP、HTTPS 等通信手段的支持（HTTPS 经常也被认为类似 SSH）。

前面已经介绍过使用如 Netmiko 模块，通过 SSH 方式替换配置文件，下面介绍如何使用 SNMP 方式来配置 Cisco IOS 设备。SNMP 一般在监控系统中作为读取设备状态信息的手段，其实 SNMP 还可以用于对设备配置的修改，这时最好使用带有安全认证的 SNMP v3 方式。

Networklore 开发了一些针对 cisco 设备的 Ansible 模块，满足 Cisco 设备 Ansible 配置的等幂性，模块软件可以从 <https://github.com/networklore/ansible-cisco-snmp/> 下载源码。这些模块依赖于 nelsnmp 0.2.2 及以上版本。当前已经包含以下模块：

- `cisco_snmp_cdp` – 全局或接口上设置 CDP 是 enable 或 disable。
- `cisco_snmp_interface` – 网络接口上配置描述、管理状态等信息。
- `cisco_snmp_portsecurity` – 启用或关闭配置端口的安全。
- `cisco_snmp_save_config` – 把 running 配置文件保存到 startup 中。
- `cisco_snmp_switchport` – 修改交换机端口模式 access/trunk 或 access/native vlan。
- `cisco_snmp_vlan` – VLAN 的创建、删除、重命名。

首先，需要在管理节点上安装 nelsnmp 软件：

```
pip install nelsnmp
```

如果后来有需要更新 nelsnmp 模块支持新的 MIB 库，只需要运行：

```
pip install nelsnmp --upgrade
```

其次，需要被管节点上配置 SNMP。

测试环境：SNMPv2，如下所示：

```
snmp-server community [write-community-string] rw [acl]
```

生产环境：SNMPv3，如下所示：

```
ip access-list standard ACL-ANSIBLE-HOST
permit host 192.168.1.100
snmp-server view V3ISO iso included
snmp-server group ANSIBLEGRP v3 priv write V3ISO
snmp-server user ansible ANSIBLEGRP v3 auth sha AuthPwdd123 priv aes
128 PrivPwdd123 access A
```

通过 SNMP 修改的只是修改 running-config，因此在使用 Ansible playbook 时需要触发 `cisco_snmp_save_config` 模块进行保存。下面是个创建 VLAN 的 playbook 实例：

```
---
- hosts: all
  connection: local
  gather_facts: no
  handlers:
    - include: handlers/main.yml

  tasks:
    - name: Create vlan 100 with SNMPv3
      cisco_snmp_vlan:
        host={{ inventory_hostname }}
        version=3
        vlan_id=100
        vlan_name=SNMPV3VLAN
        state=present
        username=snmp_v3_user
        level=authPriv
        integrity=sha
        privacy=aes
        authkey=AuthPassword123
        privkey=PrivPassword456
        notify: save config
```

handlers/main.yml 文件如下所示：

```
---
- name: save config
  cisco_snmp_save_config:
```



```

host={{ inventory_hostname }}
version=3
username=snmp_v3_user
level=authPriv
integrity=sha
privacy=aes
authkey=AuthPassword123
privkey=PrivPassword456

```

运行上述 Ansible playbook 之后，将在选择的交换机上创建 VLAN 100，如果之前交换机上没有 VLAN 100，将会触发 save\_config 模块进行保存。

## 13.5 网络设备厂商提供接口实现自动化

本节将简要介绍如何通过 Ansible 来配置 Cisco NXOS、Junos OS、Cumulus Linux 三种常见的典型网络设备，及它们需要的模块安装、功能、实例。

### 13.5.1 管理 Cisco NX-OS

Cisco 多年引领网络技术发展的潮流，近年推出了 Nexus 系列交换机，支持通过编程和自动化的部署方式，可以直接使用 Python 脚本、第三方的自动化工具（Ansible、Puppet、Chef 等）以及直接调用称为 NX-API 的应用编程接口。

NX-API 是基于 Cisco NXOS 系统的类 REST API，允许网络管理员或软件开发人员通过对网络设备的 API 发送命令行指令，指令格式可以是 JSON、XML 或 JSON-RPC。

#### 1. NX-OS 的 Ansible 模块说明

现在已经有比较完整的、超过 30 个支持 Cisco NXOS 配置的 Ansible 模块，支持配置全局参数、接口参数、高可用参数、设备管理参数等。

全局参数设置：

- nxos\_igmp\_snooping——管理 IGMP snooping 全局变量。
- nxos\_static\_routes——管理静态路由配置。
- nxos\_mtu——管理 NEXUS 交换机的 MTU 设置。

- `nxos_vtp`——管理 VTP 配置。
- `nxos_vpc`——管理全局 VPC 配置。
- `nxos_igmp`——管理 IGMP 全局配置。
- `nxos_command`——发送原始指令到 CISCO 的 NX-API 变更设备配置。
- `nxos_feature`——设置 NX-API 管理设备的特性。
- `nxos_ntp`——管理核心 NTP 配置。

#### 接口参数设置：

- `nxos_switchport`——管理二层交换接口配置。
- `nxos_vpc_interface`——管理 VPC 接口配置。
- `nxos_get_interface`——获取特定接口的详细状态信息。
- `nxos_interface`——管理接口的物理属性。
- `nxos_vrf`——管理 VRF 全局配置。
- `nxos_vrf_interface`——管理接口特定的 VRF 配置。
- `nxos_vlan`——管理 VLAN 资源与属性。
- `nxos_ipv4_interface`——管理接口三层 ipv4 属性。
- `nxos_portchannel`——管理 port-channel 接口。
- `nxos_udld`——管理 UDLD 全局配置参数。
- `nxos_udld_interface`——管理 UDLD 接口配置参数。

#### 高可用参数设置：

- `nxos_vrrp`——通过 NX-API 管理 vrrp 配置。
- `nxos_hsrp`——通过 NX-API 管理 hsrp 配置。

#### 设备管理参数设置：

- `nxos_dir`——管理 NX-OS 文件系统的目录和文件。
- `nxos_copy`——从远端服务器拷贝文件到 NEXUS 交换机。
- `nxos_save_config`——保存运行的配置文件。
- `nxos_get_facts`——通过 NX-API 获取交换机的 facts 信息。
- `nxos_get_neighbors`——从 NX-API 获取邻居 (neighbor) 详细信息。
- `nxos_ping`——从 NEXUS 交换机用 ping 测试可达性。

- nxos\_snmp\_host——管理 SNMP 主机配置。
- nxos\_snmp\_community——管理 SNMP 团体字符串配置。
- nxos\_snmp\_location——管理 SNMP 位置信息。
- nxos\_snmp\_trap——管理 SNMP 陷阱 (traps)。
- nxos\_snmp\_user——管理 SNMP 监控用户。
- nxos\_snmp\_contact——管理 SNMP 联系信息。

对每个模块更详细内容见 <https://github.com/jedelman8/nxos-ansible/blob/master/docs/nexus-module-docs.md>, 有针对每个模块详细的配置参数、使用实例、注意事项的说明。

## 2. 准备部署环境

第一步, 准备 Ansible 控制主机。

Ansible 可以安装在各种操作系统环境中, 在这里我们以安装在 Ubuntu 上为例, 或者可以直接从 Cisco DevNet 社区 (<https://developer.cisco.com/downloads/all-in-one-VM-1.3.0.181.ova>) 下载 all-in-one onePK 虚拟机。下面是准备 Ansible 控制主机的过程。

1) 更新系统, 如下所示:

```
sudo apt-get update
```

2) 安装 pip 软件包:

```
sudo apt-get install python-pip
```

3) 安装 Ansible:

```
sudo pip install ansible
```

4) 确保安装了 SSH:

```
sudo apt-get install openssh-server
```

5) 更新 /etc/hosts 文件。需要保证能够 ping 通 Nexus NX-API 交换机, 下面表示添加一台 n9k1 记录:

```
cisco@onepk:~$ cat /etc/hosts
127.0.0.1 localhost
```

```
127.0.1.1 onepk
```

```
192.168.1.133 n9k1 # <---- showing one Nexus 9000
```

现在，ping n9k1 应该能从 Nexus 交换机的管理 IP 192.168.1.133 得到响应。

第二步，安装 Cisco 的依赖软件。

安装了 Ansible 之后，就可以完全自动化管理 Linux 环境，包括对服务、软件包、文件等的管理。但是由于 Ansible 的“Core”模块没有包含能够管理、自动化网络设备的能力，这需要通过定制化集成方式来扩展功能。对于 Cisco Nexus 交换机，需要有 Python 的 xmltodict 库支持把 XML 转换成 JSON，并且需要带有 Python 的 wrappers、helper 函数来支持与 NX-API 设备通信，所有这些只要通过 pip 安装 pycsco 软件包就可以了：

```
cisco@onepk:~$ sudo pip install pycsco
```

第三步，安装 Cisco NX-OS 的 Ansible 模块。

另外，还需要安装能够自动化、管理 Nexus NX-API 系统的软件包步骤如下。

1) 安装 git:

```
cisco@onepk:~$ sudo apt-get install git
```

2) 克隆 jedelman8/nxos-ansible 软件:

```
cisco@onepk:~$ git clone https://github.com/jedelman8/nxos-ansible.git
```

3) 创建两个新目录:

```
cisco@onepk:~$ sudo mkdir -p /etc/ansible  
cisco@onepk:~$ sudo mkdir -p /usr/share/ansible
```

4) 把软件模块移到这两个目录:

```
cisco@onepk:~$ sudo mv nxos-ansible/files/ansible.cfg /etc/ansible/  
ansible.cfg  
cisco@onepk:~$ sudo mv nxos-ansible/library/ /usr/share/ansible/cisco_  
nxos
```

这么做主要有两个原因：其一，用户可以创建多个工作目录，playbook 目录就不一定总要保存在本地；其二，可以使用 ansible-doc 帮助工具支持 Cisco 模块，即提供 Ansible 模块的内置帮助功能。另外对默认的 ansible.cfg 有变化时添加设置信息收集标识 gathering = explicit，否则需要在每个 playbook 中设置 gather\_facts: False。

5) 创建、编辑授权文件。这不是必须的，但建议这么设置，在执行 playbooks 时候不需要带上用户名 / 口令：

```
cisco@onepk:~$ sudo mv nxos-ansible/files/.netauth ~/.netauth
cisco@onepk:~$ sudo vim .netauth
# the .netauth file
# make sure you input the proper creds for your device
---
cisco:
  nexus:
    username: "cisco"
    password: "cisco"
```

如果管理的交换机用户名 / 口令不一样，你需要在每个 playbook 的 task 中使用不同的认证信息。

### 3. 准备 Nexus 交换机

按照上面准备好 Ansible 控制主机后，下面需要配置 Nexus 交换机，最少需要如下步骤：1) 启用 NX-API 功能；2) Ansible 控制主机能连通 Nexus 交换机的管理口 mgmt0。在 Nexus 交换机中用 feature 命令启用 NX-API 服务，启用之后就会在 80 端口上监听。现在这些模块还只支持 http/80 端口请求，将来会支持 https/443 服务。

```
n9k1# config t
Enter configuration commands, one per line. End with CNTL/Z.
n9k1(config)# feature nxapi
n9k1(config)# exit
n9k1# show nxapi
nxapi enabled
Listen on port 80
Listen on port 443
```

针对 mgmt0 接口和对外的连接，需要先完成下面的配置：

- 把 mgmt0 配上管理 IP 地址。



- 为管理 VRF 配置一条默认路由，或最少有条能够访问 Ansible 控制主机的路由。
- 测试 Ansible 控制主机与 mgmt0 端口的连通性。

#### 4. 使用场景

下面介绍实际应用场景，需要配置的是两台汇聚交换机和 1 台接入交换机，两台 N9K 之间做基于 HSRP 的高可用，按照下面分组，即在资源清单文件 /etc/ansible/hosts 中设置：

```
[all:vars]
ansible_connection=local
```

```
[spine]
n9k1
n9k2
```

```
[leaf]
n3k1
```

对于这些配置文件，可以直接使用或进行简单修改，当需要执行 playbook 时，之间可以按照下面命令格式：

```
cisco@onepk:~/nxos-ansible$ ansible-playbook <PLAYBOOKNAME.YML> -i hosts
```

本例场景涉及的主要文件有：

- hosts：配置资源清单文件。
- site.yml：playbook 的入口。
- host\_vars：包括 n3k1.yml、n9k1.yml、n9k2.yml。
- roles：包含汇聚（spine）、接入（leaf）两个角色。

spine：tasks/main.yml，生成汇聚的配置脚本文件 vars/main.yml。

leaf：tasks/main.yml，生成接入的配置脚本文件 vars/main.yml。

site.yml 文件如下：

```
- hosts: spine
  roles:
    - role: spine
```

```

- hosts: leaf
  roles:
    - role: leaf

- hosts: all
  tasks:
    - name: save config
      nxos_save_config: host={{ inventory_hostname }}

```

host\_vars: n3k1.yml 文件如下:

```

layer3ip:
  - { interface: vlan10, ip: 10.1.1.10.10, mask: 24 }

```

host\_vars: n9k1.yml 文件如下:

```

---
layer3ip:
  - { interface: vlan10, ip: 10.1.1.10.2, mask: 24 }
  - { interface: vlan20, ip: 10.1.1.20.2, mask: 24 }

```

vpc:

```

dummy:
  domain: 100
  systempri: 2000
  rolepri: 1000
  pk1:

```

```

    src: 10.1.1.20.2
    dest: 10.1.1.20.3

```

```

    vrf: keepalive

```

hsrp\_priority: 120

host\_vars: n9k2.yml 文件如下:

```

---
layer3ip:
  - { interface: vlan10, ip: 10.1.1.10.3, mask: 24 }
  - { interface: vlan20, ip: 10.1.1.20.3, mask: 24 }

```

vpc:

```

dummy:

```

```

domain: 100
systempri: 2000
rolepri: 2000
pk1:
    src: 10.1.20.3
    dest: 10.1.20.2
    vrf: keepalive
hsrp_priority: 100

```

roles: leaf: vars/main.yml 文件如下:

```
vlan: "2-19,99"
```

```

named_leaf_vlans:
- { vlan_id: 10, name: test_segment }
- { vlan_id: 99, name: native }

```

```
uplink_interfaces:
```

```

- Ethernet1/49
- Ethernet1/50

```

```
logical_interfaces_to_create:
```

```
- vlan10
```

roles: spine: tasks/main.yml 文件如下:

```
---
# VLAN 配置
```

```
- name: ensure VLANs exist on switches
```

```

  nxos_vlan: vlan_id={{ vlans }} state=present host={{ inventory_
  hostname }}

```

```
- name: ensure names exist for important vlans
```

```

  nxos_vlan: vlan_id={{ item.vlan_id }} name={{ item.name }} host={{
  inventory_hostname }} state=present
  with_items: named_spine_vlans

```

```
# Layer 2 交换端口配置
```

```
- name: L2 config for all ports except peer keepalive
```

```

  nxos_switchport: interface={{ item }} mode=trunk native_vlan={{
  native }} trunk_vlans={{ trunk_vlans }} host={{ inventory_
  hostname }}

```

```
  with_items: default_trunk_interfaces
```

```
- name: L2 config for peer keepalive link
```

```

    nxos_switchport: interface={{ item }} mode=trunk native_vlan=99
                        trunk_vlans=20 host={{ inventory_hostname }}
    with_items: pkl_interfaces
# Portchannels 配置
- name: configure portchannels
  nxos_portchannel:
    group: "{{ item.key }}"
    members: "{{ item.value.members }}"
    mode: "{{ item.value.mode }}"
    host: "{{ inventory_hostname }}"
    with_dict: portchannels
# Layer 3 配置
- name: create logical interfaces
  nxos_interface: interface={{ item }} host={{ inventory_hostname }}
  with_items: logical_interfaces_to_create
- name: ensure VRF are created
  nxos_vrf: vrf={{ item }} host={{ inventory_hostname }}
  with_items: vrfs
- name: assign VRF to peer keepalive link interfaces
  nxos_vrf_interface: interface={{ pkl_link.interface }} vrf={{ pkl_
                        link.vrf }} host={{ inventory_hostname }}
- name: assign IP addresses
  nxos_ipv4_interface: interface={{ item.interface }} ip_addr={{ item.
                        ip }} mask={{ item.mask }} host={{ inventory_hostname }}
  with_items: layer3ip
- name: config HSRP
  nxos_hsrp: group={{ item.group }} interface={{ item.interface
                        }} vip={{ item.vip }} priority={{ hsrp_priority }} host={{
                        inventory_hostname }}
  with_items: hsrp
# 全局 VPC 配置
- name: global vpc configuration params
  nxos_vpc:
    domain={{ item.value.domain }}
    system_priority={{ item.value.systempri }}
    role_priority={{ item.value.rolepri }}
    pkl_src={{ item.value.pkl.src }}
    pkl_dest={{ item.value.pkl.dest }}
    pkl_vrf={{ item.value.pkl.vrf }}
    host={{ inventory_hostname }}
  with_dict: vpc
# Portchannel VPC 配置

```

```

- name: portchannel vpc peer link configuration
  nxos_vpc_interface: portchannel=10 peer_link=true host={{
    inventory_hostname }}
- name: portchannel vpc configuration
  nxos_vpc_interface: portchannel=11 vpc=11 host={{ inventory_hostname }}

```

roles: spine: vars/main.yml 文件如下:

```

vlangs: "2-20,99"
trunk_vlangs: "2-20"
native: 99

```

```
named_spine_vlangs:
```

```

- { vlan_id: 10, name: test_segment }
- { vlan_id: 20, name: peer_keepalive }
- { vlan_id: 99, name: native }

```

```
default_trunk_interfaces:
```

```

- Ethernet1/1
- Ethernet1/2
- Ethernet2/1

```

```
pk1_interfaces:
```

```
- Ethernet2/12
```

```
logical_interfaces_to_create:
```

```

- vlan10
- vlan20

```

```
vrfs:
```

```
-- keepalive
```

```
pk1_link:
```

```
vrf: keepalive
```

```
interface: vlan20
```

```
hsrp:
```

```

- { interface: vlan10, vip: 10.1.10.1, group: 10 }
- { interface: vlan20, vip: 10.1.20.1, group: 20 }

```

```
portchannels:
```

```
10:
```

```
members:
```

```

- Ethernet1/1
- Ethernet1/2

```

```
mode: active
```

```
11:
```

```
members:
```

```
- Ethernet2/1
```



```

    mode: active
12:
    members:
      - Ethernet2/12
    mode: active

```

roles: leaf: tasks/main.yml 文件如下:

```

# VLAN 配置
- name: ensure VLANs exist on switches
  nxos_vlan: vlan_id={{ vlans }} state=present host={{ inventory_
    hostname }}

- name: ensure names exist for important vlans
  nxos_vlan: vlan_id={{ item.vlan_id }} name={{ item.name }} host={{
    inventory_hostname }} state=present
  with_items: named_leaf_vlans
# Layer 2 交换接口配置
- name: L2 config for uplinks
  nxos_switchport: interface={{ item }} mode=trunk native_vlan=99
    trunk_vlans=2-19 host={{ inventory_hostname }}
  with_items: uplink_interfaces
# Portchannels 配置
- name: portchannel for uplinks
  nxos_portchannel:
    group: 100
    members: "{{ uplink_interfaces }}"
    mode: active
    host: "{{ inventory_hostname }}"
    state: present
# Layer 3 配置
- name: create logical vlan interfaces
  nxos_interface: interface={{ item }} host={{ inventory_hostname }}
  with_items: logical_interfaces_to_create

- name: assign IP addresses
  nxos_ipv4_interface: interface={{ item.interface }} ip_addr={{ item.
    ip }} mask={{ item.mask }} host={{ inventory_hostname }}
  with_items: layer3ip

```

### 13.5.2 管理 JUNOS

Juniper 提供支持 Ansible 管理 JUNOS 设备的函数库, 在 Ansible 官方的 Galaxy

网站中维护，这些函数库使用方便简单、功能强大，是操作、配置运行有 JUNOS 设备的有效手段。包括安装、升级 JUNOS，部署网络中的配置项，加载配置文件，查询信息，管理设备的重置、重启、关闭。是在网络中快速部署 Juniper 网络设备的强大工具。

虽然 Ansible 通常需要被管节点支持 Python，但在管理运行 JUNOS 的设备时是不需要的。执行支持 JUNOS 的 Ansible 模块时是基于 SSH 会话之上调用 netconf，因此需要确保被管理的 JUNOS 设备上已经开启了 netconf 功能。更多有关支持 Ansible 的 JUNOS 函数库的详细信息，请参考如下资源：

- 支持 Ansible 的 JUNOS 函数库，托管在 GitHub 源码仓库中，即 <https://github.com/Juniper/ansible-junos-stdlib/>。
- 支持 Ansible 的 JUNOS 模块文档见 <http://junos-ansible-modules.readthedocs.org/>。
- 支持 Ansible 的 JUNOS 角色在 <https://galaxy.ansible.com/> 网站。
- 关于 Ansible for JUNOS 在 Juniper 官方网站有详细信息，即 [http://www.juniper.net/techpubs/en\\_US/release-independent/junos-ansible/information-products/pathway-pages/index.html](http://www.juniper.net/techpubs/en_US/release-independent/junos-ansible/information-products/pathway-pages/index.html)。

当前已经支持的模块有：

- `junos_commit`——提交设备上的配置信息。
- `junos_get_config`——查询、获取 JUNOS 设备的配置信息。
- `junos_get_facts`——查询、获取主机的信息。
- `junos_install_config`——修改运行 JUNOS 的设备的配置文件。
- `junos_install_os`——安装 JUNOS 软件包。
- `junos_rollback`——回滚设备的配置信息。
- `junos_shutdown`——关闭或重启运行 JUNOS 的设备。
- `junos_srx_cluster`——Enable/DisableSRX 设备的 Cluster 功能。
- `junos_zeroize`——初始化成出厂配置，包括路由引擎。

下面详细介绍如何使用支持 JUNOS 的 Ansible 模块来创建管理 Juniper 设备的 playbook，并对执行过程和结果进行说明。

## 1. 准备 Ansible 管理 JUNOS 环境

要使用 Ansible 管理 JUNOS 设备，需要在 Ansible 管理控制主机上安装 `Juniper.junos` 角色，以及 `Juniper.junos` 的依赖软件包 `ncclient` 和 `junos-eznc`。具体过程如下：

```
[root@ansiblecontrol ~]# pip install ncclient
[root@ansiblecontrol ~]# pip install junos-eznc
[root@ansiblecontrol ~]# ansible-galaxy install Juniper.junos
- downloading role 'junos', owned by Juniper
- downloading role from https://github.com/Juniper/ansible-junos-stdlib/
  archive/1.2.0.tar.gz
- extracting Juniper.junos to /etc/ansible/roles/Juniper.junos
- Juniper.junos was installed successfully
```

同时需要在被管理的 JUNOS 设备上开启 `netconf` 功能：

```
# set system services netconf ssh
```

## 2. 创建支持 JUNOS 的 playbook

在安装好支持 JUNOS 的 Ansible 模块之后，需要编写管理 JUNOS 设备的 playbook，创建一个带 `.yaml` 扩展名的 playbook 文件。

1) playbook 基本信息如下：

```
---
- name: Get Device Facts
  hosts: dc1
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
```

在这部分内容中，说明如下：

- 通过 `name` 对 `playbook` 进行说明描述。
- 定义将要运行这个模块的主机或主机组，这些主机或主机组在资源清单文件中进行描述，这里是主机 `dc1`。
- 在 `playbook` 中引用 `Juniper.junos` 角色，在后面的 `task` 中就可以引用 JUNOS 中实现的功能。

- 由于运行 JUNOS 系统的设备不能直接支持 Python，因此 playbook 将在管理控制主机上调用本地 Python 环境执行，即 `connection: local`。
- 由于 Ansible 在管理控制主机上执行，为了避免收集到大量的 facts 信息，建议设置 `gather_facts: no`，当然这不是必须的。

2) 定义任务，任务可以是一个或多个。下面是检测运行 JUNOS 设备的 `netconf` 可连接的脚本。连接 830 端口，超时 5 秒钟：

```
tasks:
- name: Checking NETCONF connectivity
  wait_for: host={{ inventory_hostname }} port=830 timeout=5
```

### 3. 执行管理 JUNOS 的实例

通过上面的介绍后，我们给出一个完整的 playbook 实例：

```
---
- name: Get Device Facts
  hosts: dc1
  roles:
- Juniper.junos
  connection: local
  gather_facts: no

tasks:
- name: Checking NETCONF connectivity
  wait_for: host={{ inventory_hostname }} port=830 timeout=5
- name: Retrieve information from devices running Junos OS
  junos_get_facts:
    host={{ inventory_hostname }}
    register: junos
- name: version
debug: msg="{{ junos.facts.version }}"
```

这个 playbook 包括连接检测、收集 facts 信息、显示 JUNOS 软件版本。

在管理控制主机上执行 `ansible-playbook` 命令，将会得到如下结果：

```
[root@ansible-cm]# ansible-playbook playbook.yml
```

```

PLAY [Get Device Facts] *****

TASK: [Checking NETCONF connectivity] *****
ok: [dc1a.example.com]

TASK: [Retrieve information from devices running Junos OS]
*****
ok: [dc1a.example.com]

TASK: [version] *****
ok: [dc1a.example.com] => {
  "msg": "14.1R1.10"
}

PLAY RECAP *****
dc1a.example.com      : ok=3    changed=0    unreachable=0
failed=0

```

### 13.5.3 管理 Cumulus Linux

随着软件定义数据中心（SDDC）和软件定义存储（SDS）的趋势愈演愈烈，传统连网设备、存储设备和服务器之间的界限变得越来越模糊。鉴于这些以前各自为政的领域已经开始慢慢地融合，白盒交换机被再次提起，正在开始获得越来越多的关注。Cumulus Networks 和 Big Switch Networks 等供应商积极投身于该创新领域。

Cumulus Linux 是一款面向物理层面、为连网硬件而设计的 Linux 操作系统，可以利用它调整数据中心的规模，更好地实现网络自动化管理。它硬件兼容性很强，支持传统和现代网络架构。它将操作系统从连网设备中分离出来，这是大规模、面向软件定义数据中心的一个关键组成部分，也是提高架构灵活性的核心。利用现有的 Linux 核心数据结构完成与开源 / 商业 Linux 应用的整合，当前已经有大量的软件定义连网组件可在 Cumulus Linux 上运行。

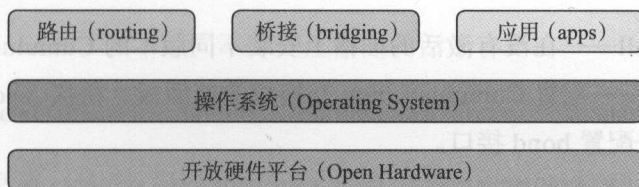


图 13-3 Cumulus Linux 框架



Cumulus 正专注于让网络实现超高容量互连，并且易于部署、扩展、容易操作，整合了开源与客户所使用的大量最新网络技术。网络工程师像管理服务器一样管理运行着 Cumulus Linux 网络操作系统的网络设备，将网络设备看作是另一种服务的 Linux 服务器。可以用与管理 Linux 服务器共享相同的团队、工具和流程去管理 Cumulus 白盒交换机，将以前特殊的网络环境配置方式变成一种通用方式。Cumulus 白盒交换机是将来自大型传统交换机供应商的私有交换机接口替换成 Linux 服务器管理员都能理解的通用 Linux 接口。采用与 Linux 服务器自动配置安装过程的相同方法，除了一些小更新和修改之外，完全不需要更多的特殊考虑，Cumulus Linux 控制器会自动配置其白盒交换机安装设置，这个网络配置包括客户 Pod（云基础环境中的一个可用域）和用于管理的命令来控制 Pod。

虽然网络工程师现在可以通过自动化、CLI（命令行接口）和 API 编程控制管理交换机，但 Cumulus Linux 的特别之处是它的 CLI 和 API 与每一位系统工程师和云服务工程师长期使用的 Linux 工具完全相同，如路由器命令和 `ipconfig` 等。完全不需要学习新东西，所有东西都由硬件加速。

Linux 服务器与运行 Cumulus Linux 的交换机之间并没有很大的差别——主要差别就是每一台设备上的端口数量。Linux 服务器可能只有 2 ~ 4 个 2 层和 3 层连接以太网端口，而 Cumulus 白盒交换机则可以有 48 个 10GB 端口。使用 DevOps 自动化工具可以实现服务器与网络管理。例如，每一个交换机都运行着版本 2 和版本 3 的开放最短路径优先（Open Shortest Path First）协议，那么从配置到故障修复等所有网络操作都变得非常简单。

下面介绍如何使用 Ansible 模块管理 Cumulus Linux 网络服务。使用 Cumulus Linux 模块将会大大简化 `playbook` 的复杂性，也不需要 Jinja 模板。支持管理 Cumulus Linux 网络服务的 Ansible 模块源码可以从 <https://github.com/CumulusNetworks/cumulus-linux-ansible-modules> 下载。当前已经支持的模块有：

- `cl_img_install`——在没有激活的插槽上安装不同版本的 Cumulus Linux。
- `cl_interface`——配置 Cumulus Linux 交换机的前面板、桥接、bond 接口。
- `cl_bond`——配置 bond 接口。
- `cl_bridge`——配置桥接接口。
- `cl_interface_policy`——配置接口强制策略。

- `cl_ports`——配置接口属性，如在 `/etc/cumulus/ports.conf` 中定义的 Breakout 端口。
- `cl_license`——加载 Cumulus Linux 授权，默认支持 2 个网络接口 `sw1`、`sw2`。
- `cl_quagga_ospf`——配置基础的 OSPFv2 全局参数和 OSPFv2 接口参数。
- `cl_quagga_protocol`——在 Cumulus Linux 上启用 Quagga 服务。

### 1. 准备管理 Cumulus Linux 环境

准备管理 Cumulus Linux 环境首先需要一台支持 Ansible 模块的管理控制主机。需要在 Ansible 控制主机上安装 `cumulus.CumulusLinux` 角色，包含支持管理 CumulusLinux 的模块，可以直接从 Ansible 官方的 Galaxy 网站安装，即用 `ansible-galaxy install cumulus.CumulusLinux` 命令，如下所示：

```
[root@switch]# ansible-galaxy install cumulus.CumulusLinux
- downloading role 'CumulusLinux', owned by cumulus
- downloading role from https://github.com/cumulusnetworks/cumulus-
  linux-ansible-modules/archive/2.2.1.tar.gz
- extracting cumulus.CumulusLinux to /etc/ansible/roles/cumulus.
  CumulusLinux
- cumulus.CumulusLinux was installed successfully
[root@switch]#
```

设置 Ansible 函数库的环境变量，通常 Ansible 的环境参数在 `/etc/ansible/ansible.cfg` 中配置。当然也可以根据 Ansible 对配置文件的检查顺序在当前目录下直接创建 `ansible.cfg`，这个配置不会影响其他用户。下面是对 `cumulus.CumulusLinux` 库函数的设置：

```
[root@switch]# cat ansible.cfg
[defaults]
library=/etc/ansible/roles/cumulus.CumulusLinux/library/:/usr/share/
ansible
host_key_checking=False
hostfile = ansible.hosts
```

其中：

- `library = /etc/ansible/roles/cumulus.CumulusLinux/library/:/usr/share/ansible`

安装完成 `cumulus.CumulusLinux` 角色之后，Ansible 控制主机会把 Cumulus Linux 模块安装在 `/etc/ansible/roles/cumulus.CumulusLinux/library/`；，这里分号 (:) 相

当于与 (AND) 的意思, `/usr/share/ansible` 是 Ansible 模块默认的安装位置。这行就是说明指定 Cumulus Linux 模块位置和通常 Ansible 模块的位置。

- `host_key_checking=False`

默认是启用检查 SSH 密钥的, 有时这可能不如直接用口令来得便捷。

- `hostfile = ansible.hosts`

查找的资源清单文件在当前目录的 `ansible.hosts`, 而不是默认的 `/etc/ansible/hosts`。

## 2. 管理 Cumulus Linux 实例

下面以 CumulusNetworks 官方提供的 `cumulus-linux-ansible-modules` 模块中的实例看看如何管理 Cumulus Linux 系统。

在上一节中已经创建了管理的控制主机的环境, 下面就在 `ansible.hosts` 中添加需要管理的交换机, 如:

```
[root@ansiblecontrol playbook]# cat ansible.hosts
[switches]
    cumulus
```

其中 `cumulus` 是我们实例中管理的 Cumulus Linux 交换机, 本例中的管理 IP 是 192.168.1.10/24。

然后编写管理的 playbook 脚本 `default.yml`, 包含了配置端口的 `task/interfaces.yml` 和变更之后需要处理的动作 `handlers/main.yml`:

```
[root@ansiblecontrol playbook]# cat default.yml
---
- hosts: all
  tasks:
    - include: tasks/interfaces.yml
  handlers:
    - include: handlers/main.yml
```

配置端口的脚本中调用了网络接口配置文件 `files/interface`, 然后配置 `lo`、`eth0`、`swp1`、`swp2` 端口:

```
[root@ansiblecontrol playbook]# cat tasks/interfaces.yml
```

```

- name: Overwrite interfaces file
  copy: dest=/etc/network/interfaces src=files/interfaces

- name: Configure lo
  cl_interface: name=lo addr_method=loopback
  notify: reload networking

- name: Configure eth0
  cl_interface: name=eth0 addr_method=dhcp
  notify: reload networking

# With all defaults
- name: Configure interface with defaults
  cl_interface: name=swp1
  notify: reload networking

# Over-ride defaults
- name: Configure interface without defaults
  cl_interface:
    name: swp2
    ipv4: ['192.168.200.1']
    speed: '1000'
    mtu: 9000
    vids: ['1-4094']
    pvid: 1
    alias_name: 'interface swp2'
    virtual_mac: '11:22:33:44:55:66'
    virtual_ip: '192.168.10.1'
    mstpctl_portnetwork: true
    mstpctl_bpdguard: true
  notify: reload networking

```

处理 notify 的 reload networking 动作的 handlers/main.yml，如下所示：

```

[root@ansiblecontrol playbook]# cat handlers/main.yml
- name: reload networking
  service: name=networking state=reloaded

```

通过 ansible-playbook 执行 default.yml 脚本，如下所示：

```

[root@ansiblecontrol playbook]# ansible-playbook default.yml
PLAY [all] *****

```

```

GATHERING FACTS *****
ok: [cumulus]

TASK: [Overwrite interfaces file] *****
changed: [cumulus]

TASK: [Configure lo] *****
changed: [cumulus]

TASK: [Configure eth0] *****
changed: [cumulus]

TASK: [Configure interface with defaults] *****
changed: [cumulus]

TASK: [Configure interface without defaults] *****
changed: [cumulus]

NOTIFIED: [reload networking] *****
changed: [cumulus]

PLAY RECAP *****
cumulus                : ok=7changed=6    unreachable=0    failed=0

```

登录 Cumulus Linux 系统，查看 /etc/network/interfaces 文件，发现已经配置成之前定义的 files/interfaces 文件的内容：

```

root@cumulus:~/.ssh# cat /etc/network/interfaces
source /etc/network/interfaces.d/*

```

通过 ifconfig 可以看到 swp2-v0 的 IP 地址已经设置成要求的 192.168.10.1：

```

root@cumulus:~/.ssh# ifconfig
.....
swp2-v0  Link encap:Ethernet  HWaddr 9e:e6:ff:21:cf:0d
          inet addr:192.168.10.1  Bcast:0.0.0.0  Mask:255.255.255.255
          inet6 addr: fe80::9ce6:ffff:fe21:cf0d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
.....

```



## 13.6 本章小结

本章首先介绍现有常见的网络管理手段，包括 CLI、SNMP、WEB、NETCONF 等；然后通过 Ansible 的模板功能生成网络设备的配置文件，并由 Paramiko 的 SSH 连接类库加载运行；之后介绍由 Ansible 另一种模块调用，通过传统的 SNMP 网管协议 set 方式管理、配置网络设备；最后分别介绍 Cisco Nexus、Junos OS、Cumulus Linux 三家有代表性的网络设备，通过 Ansible 调用设备的管理 API 进行自动化管理。

网络配置和管理正处于变革的初始阶段，随着 SDN 的迅猛发展，网络系统将会越来越开放，网络厂家也将会提供越来越完善的 API 服务功能，大量的网络设备将集成到现有的 DevOps 自动化工具中来，为大规模、自动化管理网络系统管理奠定了坚实的基础。

## Ansible API

在前面的章节中，我们已经对 Ansible 有了较全面的认识，也通过一些案例介绍了它的应用场景。本章介绍 Ansible 的两个常用 API，之后我们用 API 对 Ansible 进行 2 次封装或者做成单独的系统对外提供 HTTP 接口的服务，当然也可以使用 API 与公司现有的其他系统进行整合，等等。前面我们也介绍了 Ansible 的日常模块使用 Ad-Hoc 模式与配置管理 playbooks 两大核心功能，也通过一些案例去企业中真正地使用它。本章将分别针对 Ansible 的两大核心功能的 API 进行讲解，以便更好地扩展 Ansible。然后通过实例讲解如何使用 Flask Web 框架去封装 Ansible API。最后简单介绍两个关于编写 Ansible Web 管理系统需要涉及的分布式异步任务工具 Celery 和 jQuery。

Ansible 的扩展性非常强，前面我们也通过一些语言去定义一些自己的 module。Ansible 更是支持使用 Python 语言编写它的各种插件，当然我们平常使用最频繁的 Ansible 的 Ad-Hoc 模式以及 playbooks 操作也提供相应的 Python API，这样我们就可以使用 Python 语言去直接操作 Ansible 了。下面我们开始讲解 Ansible 的 runner 和 playbooks API 内容。当然 Ansible 还有其他的 API，本章就不进行相应介绍了，有一定 Python 开发能力的读者建议去阅读 Ansible 源码。

### 14.1 runner API

我们平常使用 Ansible 的时候经常会使用到它的 Ad-hoc 模式。在 Ansible 的 Ad-

Hoc 模式下的所有操作我们可以使用 Ansible 的 runner API 进行操作，所以 runner API 就能实现我们日常所有的 Ad-Hoc 操作。下面就来看一看 runner API。默认 runner API 所有的代码在当前 Python 版本的 `site-packages/ansible/runner` 目录下。以下我们用 shell 模块在对比下 runner API 的使用。为了更好地演示，我的代码直接运行在 ipython 环境下；为了更好地分析 API 返回的数据结果，我的目标主机中会有一台连接失败的机器：

```
[root@python ~]# ansible all -m shell -a 'hostname'
192.168.1.116 | success | rc=0 >>
Master

192.168.1.117 | success | rc=0 >>
Minion

192.168.1.118 | success | rc=0 >>
python

192.168.1.119 | FAILED => SSH Error: ssh: connect to host 192.168.1.119
port 22: No route to host
while connecting to 192.168.1.119:22
It is sometimes useful to re-run the command using -vvvv, which prints
SSH debug output to help diagnose the issue.
```

这是我们日常执行 shell 模块的使用方法。下面来看一下通过 runner API 是怎么实现的 (ipython 环境)：

```
In [28]: import ansible.runner

In [29]: runner=ansible.runner.Runner(module_name='shell',module_
args='hostname',pattern='all', forks=10)

In [30]: result=runner.run()

In [31]: for host,ret in result['contacted'].items():
.....:     print host,ret['stdout']
.....:
192.168.1.117 Minion
192.168.1.116 Master
192.168.1.118 python

In [32]: for host,ret in result['dark'].items():
```

$\vdots$ 

```
while connecting to 192.168.1.119:22
```

要查看 runner 方法的参数，可以使用 `help (ansible.runner.Runner)`，或者查看

$u'_{delta'}: u'0:00:00.006049'$ ,

```

    u'end': u'2015-05-20 00:10:48.462575',
    'invocation': {'module_args': u'hostname',
                   'module_name': 'shell'},
    u'rc': 0,
    u'start': u'2015-05-20 00:10:48.456526',
    u'stderr': u'',
    u'stdout': u'python',
    u'warnings': []}},
'dark': {'192.168.1.119': {'failed': True,
                           'msg': 'SSH Error: ssh: connect to host
192.168.1.119 port 22: No route to host\n
while connecting to 192.168.1.119:22\nIt is
sometimes useful to re-run the command using
-vvvv, which prints SSH debug output to help
diagnose the issue.'}}}
```

contacted 里面存储着所有执行成功的信息，dark 里面存储着所有执行失败的信息。通过这个数据结构可以很清晰地看出整个 runner API 的结果，然后可以在这个结果里面挑选我们关心的数据。我们可以使用 Python 语言去封装 Ansible 的 runner API，或者把我们经常使用的一些模块封装成脚本的形式运行，这样就非常方便地扩展 Ansilbe 的相关功能了。

## 14.2 playbook API

接下来我们来了解 playbook API。平时我们编写好 playbook 之后都是通过 ansible-playbook 命令去运行的。相比之下，playbook API 比 runner API 要复杂些，因为我们执行 playbook 的时候需要引入一些其他的插件，比如前面介绍 callback 插件。默认 palybook 所有的代码存放在当前 Python 版本的 site-packages/ansible/playbook 下。下面我们通过手动执行 ansible-palybook 和使用 palybook API 来执行一个 playbook。首先我们来看一下 playbook：

```
[root@python ~]# cat key.yaml
```

```

---
- hosts: "{{hosts}}"
  gather_facts: false
  tasks:
```



```

- name: key
  authorized_key: user=root key="{ { lookup('file', '/
root/.ssh/id_rsa.pub') } }"

```

这是一个很简单的同步 root 用户公钥的 playbook:

```
[root@python ~]# ansible-playbook -i hosts key.yaml -e "hosts=all"
```

```
PLAY [all] *****
```

```
TASK: [key] *****
```

```
ok: [192.168.1.117]
```

```
ok: [192.168.1.116]
```

```
ok: [192.168.1.118]
```

```
fatal: [192.168.1.119] => SSH Error: ssh: connect to host 192.168.1.119
port 22: No route to host
while connecting to 192.168.1.119:22
```

It is sometimes useful to re-run the command using -vvvv, which prints SSH debug output to help diagnose the issue.

```
PLAY RECAP *****
```

```
to retry, use: --limit @/root/key.yaml.retry
```

192.168.1.116	: ok=1	changed=0	unreachable=0	failed=0
192.168.1.117	: ok=1	changed=0	unreachable=0	failed=0
192.168.1.118	: ok=1	changed=0	unreachable=0	failed=0
192.168.1.119	: ok=0	changed=0	unreachable=1	failed=0

下面我们再通过 playbook API 的方式来引用和运行上面定义的 playbook，当然需要引入一些 Ansible playbook 相关的模块（ipython 环境）：

```
In [1]: from ansible.inventory import Inventory
```

```
In [2]: from ansible.playbook import PlayBook
```

```
In [3]: from ansible import callbacks
```

```
In [4]: inventory = Inventory('./hosts')
```

```
In [5]: stats = callbacks.AggregateStats()
```

```
In [6]: playbook_cb = callbacks.PlaybookCallbacks()
```

```

In [7]: runner_cb = callbacks.PlaybookRunnerCallbacks(stats)

In [9]: results = PlayBook(playbook='key.yaml', callbacks=playbook_
      cb, runner_callbacks=runner_cb, stats=stats, inventory=inventory, ext
      ra_vars={'hosts': 'all'})

In [10]: res = results.run()

PLAY [all] *****

TASK: [key] *****
ok: [192.168.1.118]
fatal: [192.168.1.119] => SSH Error: ssh: connect to host 192.168.1.119
      port 22: No route to host
      while connecting to 192.168.1.119:22
It is sometimes useful to re-run the command using -vvvv, which prints
      SSH debug output to help diagnose the issue.
ok: [192.168.1.116]
ok: [192.168.1.117]

In [11]: import pprint

In [12]: pprint.pprint(res)
{'192.168.1.116': {'changed': 0,
                  'failures': 0,
                  'ok': 1,
                  'skipped': 0,
                  'unreachable': 0},
 '192.168.1.117': {'changed': 0,
                  'failures': 0,
                  'ok': 1,
                  'skipped': 0,
                  'unreachable': 0},
 '192.168.1.118': {'changed': 0,
                  'failures': 0,
                  'ok': 1,
                  'skipped': 0,
                  'unreachable': 0},
 '192.168.1.119': {'changed': 0,
                  'failures': 0,
                  'ok': 0,
                  'skipped': 0,

```

```
'unreachable': 1}}}
```

- In [1] 表示引入 Inventory 类。
- In [2] 表示引入 PlayBook 类。
- In [3] 表示引入 callbacks 类。
- In [4] 表示定义 inventory 文件，如果是动态 inventory 指定脚本即可。
- In [5]-In[7] 表示加载 callbacks 插件。
- In [9] 表示初始化 playbook 参数。
- In [10] 表示运行 playbook，将结果保存到 res 对象内。
- In [11] 为了很好的打印 res 对象内的数据，这里引入了 pprint 模块。

关于 PlayBook 这个 class 的使用参数，我们可以使用 `help(PlayBook)` 查看或者查看 `playbook/__init__.py` 代码的 PlayBook 类下的 `__init__` 函数的参数。里面提供很多参数供我们使用。关于 playbook API 的结果相比 runner API 的结果要好理解得多，在 playbook API 的执行结果里面只会显示每台机器的状态，比如执行成功了多少个 task，执行失败了多少个 task 信息。

## 14.3 使用 Flask 封装 Ansible API

前面已经介绍了 Ansible 的 runner 和 palybook 两个 API，尽管这些 API 功能很强大，但是这些 API 功能我们不能远程调用。我们熟悉 Ansible API 后可以使用一些 Python Web 框架进行 2 次封装，方便我们远程调用。或者开发一个适合自己业务的 Ansible Web 平台等。下面我就通过一个简单的例子使用 Flask Web 框架去封装 Ansible 的 runner 和 playbooks API。关于 Flask Web 框架的知识，可以通过 Flask 官网 (<http://flask.pocoo.org/>) 进行学习，本书不进行更多的介绍。下面我们通过 Flask 对外提供 HTTP API 接口，这样就可以通过 HTTP 请求的方式操作 Ansible。Flask 代码如下：

```
#!/usr/bin/python
#coding:utf-8
from ansible.inventory import Inventory
from ansible.playbook import PlayBook
from ansible import callbacks
import ansible.runner
from flask import Flask,request,jsonify,render_template,abort
```

```

import commands,json
app = Flask(__name__)

hostfile='./hosts'
'''
    http://127.0.0.1:5000/API/Ansible/playbook?ip=2.2.2.2&palybook=test
'''
@app.route('/API/Ansible/playbook')
def Playbook():
    vars={}
    inventory = Inventory(hostfile)
    stats = callbacks.AggregateStats()
    playbook_cb =callbacks.PlaybookCallbacks()
    runner_cb =callbacks.PlaybookRunnerCallbacks(stats)
    hosts=request.args.get('ip')
    task=request.args.get('playbook')
    vars['hosts'] = hosts
    play=task + '.yaml'
    results = PlayBook(playbook=play,callbacks=playbook_cb,runner_
        callbacks=runner_cb,stats=stats,inventory=inventory,extra_vars=vars)
    res = results.run()
    return json.dumps(res,indent=4)

'''
curl -H "Content-Type: application/json" -X POST -d '{"ip":"1.1.1.1
", "module":"shell", "args":"ls -l"}' http://127.0.0.1:5000/API/
Ansible/runner
'''
@app.route('/API/Ansible/runner',methods=['POST'])
def Runner():
    print request.json
    if not request.json or not 'ip' in request.json or not 'module' in
        request.json or not 'args' in request.json:
        abort(400)
    hosts=request.json['ip']
    module = request.json['module']
    args=request.json['args']
    runner = ansible.runner.Runner(module_name=module,module_args=args,
        pattern=hosts,forks=10,host_list=hostfile)
    tasks=runner.run()
    cpis={}
    cpisl={}
    for (hostname, result) in tasks['contacted'].items():

```

```

    if not 'failed' in result:
        cpis[hostname] = result['stdout']
    for (hostname, result) in tasks['dark'].items():
        cpisl[hostname] = result['msg']
    return render_template('cpis.html', cpis=cpis, cpisl=cpisl)

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0')

```

这里简单介绍一下，首先定义了两个 `app.route`，其实就是两个 URL，一个是调用 runner API 的，一个是调用 playbook API 的其次通过 URL 的方式传入一些参数，Flask 接收到这些参数后传入 Ansible API，最后把 API 执行的结果返回给用户。关于 runner API，需要用户发起 HTTP POST 请求，然后在 POST 的数据里面携带一些 module 名称和 module 参数信息，以及目标机器。runner API 的 return 格式简单定义了一个 jinja2 模板，这个模板可以根据自己的需求去定义：

```

##### 成功设备列表 #####
{% for x,y in cpis.items() %}
===== 主机 {{ x }}=====
{{ y }}
{% endfor %}
##### 失败设备列表 #####
{% for a,b in cpisl.items() %}
===== 主机 {{ a }}=====
{{ b }}
{% endfor %}

```

下面我们就可以使用 HTTP 请求的方式操作 Ansible 了，比如通过 HTTP GET 方式向 Flask 发送一个调用 Ansible PlayBook API 的操作。针对所有主机运行 `key.yml` playbook：

```

[root@python ~]# curl "http://127.0.0.1:5000/API/Ansible/
playbook?ip=all&playbook=key"
{
  "192.168.1.117": {
    "unreachable": 0,
    "skipped": 0,
    "ok": 1,
    "changed": 0,

```



```

    "failures": 0
  },
  "192.168.1.116": {
    "unreachable": 0,
    "skipped": 0,
    "ok": 1,
    "changed": 0,
    "failures": 0
  },
  "192.168.1.119": {
    "unreachable": 1,
    "skipped": 0,
    "ok": 0,
    "changed": 0,
    "failures": 0
  },
  "192.168.1.118": {
    "unreachable": 0,
    "skipped": 0,
    "ok": 1,
    "changed": 0,
    "failures": 0
  }
}

```

执行完成后可以直接显示每台主机 Playbook 中 tasks 的状态信息了。同样我们还可以使用 HTTP POST 的方式给 Ansible runner API 发送模块的执行请求，比如远程调用 Ansible 的 shell 模块，针对所有主机执行 hostname 参数：

```

[root@python ~]# curl -H "Content-Type: application/json" -X
POST -d '{"ip":"all","module":"shell","args":"hostname"}'
http://127.0.0.1:5000/API/Ansible/runner
##### 成功设备列表 #####

```

```

===== 主机 192.168.1.117=====
Minion

```

```

===== 主机 192.168.1.116=====
Master

```

```

===== 主机 192.168.1.118=====

```

```
python
##### 失败设备列表 #####

===== 主机 192.168.1.119=====
SSH Error: ssh: connect to host 192.168.1.119 port 22: No route to host
while connecting to 192.168.1.119:22
It is sometimes useful to re-run the command using -vvvv, which prints
SSH debug output to help diagnose the issue.
```

因为我对 runner API 的执行结果用 Jinja 模板进行了格式化，所以我们可以很直观地看到每台机器的执行结果信息。

这就是一个最简单的使用 Flask 实现 Ansible HTTP API 的示例，当然还可以使用其他框架进行封装，大家可以根据自己的需求去定制一些 Python 脚本等。这里主要是为了大家能理解 Ansible 的 runner API 和 playbook API 后能够进行一些简单的封装以及二次开发等。如果有 Python 项目开发经验的读者，可以使用其他框架编写一个 Ansible Web 管理系统。与 Ansible 之间的操作直接使用 runner API 和 playbook API 即可。当然还需要考虑其他因素，比如目前 Ansible 的 API 在调用的时候都是阻塞的，如果你执行一个 playbook 时间很长，可能会出现前端用户一直处于等待状态，甚至超时等等。这个时候我们可以引入一些其他工具去执行，比如 Celery 这类的异步任务项目以及 Web 前端的 Ajax 技术等。

## 14.4 使用 Celery 实现任务异步化

虽然在上一节我们实现了远程调用 Ansible API，但是细心的读者会发现，Ansible 的 API 默认本身是阻塞的，例如我们在请求 Ansible API 的时候，如果 Ansible 本身没有完成这个工作，用户那边将一直处于等待状态。其实默认这样的应用对网民的体验感不是很好，下面我们来简单整合 Ansible playbook API 和 Celery。关于 Celery 的相关介绍，本书不多讲，大家可以通过 Celery 官网进行了解和学习，只要明白 Celery 是一个实现分布式异步任务的工具即可，它能把一些执行等待时间很长的任务进行异步化。下面的例子我是使用 Redis 当作 Celery 的 broker，读者可根据情况挑选自己熟悉的 broker。我们来看一下 Flask 的代码：

```
from celery import Celery
```

```

import json
from flask import Flask, abort, jsonify, request, session
from ansible.inventory import Inventory
from ansible.playbook import PlayBook
from ansible import callbacks
import jinja2
from tempfile import NamedTemporaryFile

app = Flask(__name__)
app.config['SECRET_KEY'] = 'top-secret!'
app.config['CELERY_BROKER_URL'] = 'redis://localhost:6379/0'
app.config['CELERY_RESULT_BACKEND'] = 'redis://localhost:6379/0'
celery = Celery(app.name, broker=app.config['CELERY_BROKER_URL'])
celery.conf.update(app.config)

@celery.task
def adduser(ips, users):
    inventory = """
    {% for i in hosts -%}
    {{ i }}
    {% endfor %}
    """

    inventory_template = jinja2.Template(inventory)
    rendered_inventory = inventory_template.render({'hosts': ips})
    hosts = NamedTemporaryFile(delete=False, suffix='.tmp', dir='/tmp/
        ansible/')
    hosts.write(rendered_inventory)
    hosts.close()
    inventory = Inventory(hosts.name)
    stats = callbacks.AggregateStats()
    playbook_cb = callbacks.PlaybookCallbacks()
    runner_cb = callbacks.PlaybookRunnerCallbacks(stats)
    vars={}
    vars['users'] = users
    results = PlayBook(playbook='user.yaml', callbacks=playbook_
        cb, runner_callbacks=runner_cb, stats=stats, inventory=inventory, e
        xtra_vars=vars)
    res = results.run()
    logs = []

```

```

logs.append("finish playbook\n")
logs.append(str(res))
return logs

@app.route('/', methods=['GET', 'POST'])
def index():
    return render_template('index.html')

@app.route("/add", methods=['POST'])
def one():
    ips = [ i.encode('ascii') for i in request.form.getlist('ips') ]
    users = [ i.encode('ascii') for i in request.form.getlist('users') ]
    res = adduser.apply_async((ips, users))
    context = {"id": res.task_id, "ips": ips, "users": users}
    result = "add((ips){0}, (users){1})".format(context['ips'],
        context['users'])
    goto = "{0}".format(context['id'])
    return jsonify(result=result, goto=goto)

@app.route("/add/result/<task_id>")
def show_add_result(task_id):
    task = adduser.AsyncResult(task_id)
    if task.state == 'PENDING':
        response = {
            'state': task.state,
            'status': 'Pending...'
        }
    elif task.state != 'FAILURE':
        response = {
            'state': task.state,
            'status': task.info
        }
        if 'result' in task.info:
            response['result'] = task.info['result']
    else:
        response = {
            'state': task.state,
            'status': task.info,
        }

```

```

return jsonify(response)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000, debug=True)

```

在这个 Flask 脚本里面，我只定义了一个简单的 adduser 的 Celery tasks，这个任务就是把 Ansible playbook API 操作放入 Celery 里面。当然 Celery 也支持定义很多其他 task。inventory 在这里不是指定静态或者动态的文件，是通过客户端使用 HTTP 形式 POST 的主机列表然后采用 tempfile 和 jinja2 模板动态生成每次 POST 过来的主机，这样可以更加灵活地处理目标主机。然后针对这些主机使用 Ansible playbook API 进行处理。其实这个例子就是执行 user.yaml。当然大家可以提前编写好 playbook，通过不同的参数调用不同的 playbook 任务即可。

因为 Ansible playbook API 是一个阻塞的过程，所以把这个操作放到 Celery tasks 里面了。当客户端 POST 一个任务来后，能马上响应客户端请求并且返回 Celery tasks id，客户端可以根据这个 id，查看这个请求的实时状态。如果查询 Celery 中该 tasks id 的状态运行完成了，这个时候直接把运行结果返回给用户。下面我们运行代码，首先需要启动 Celery 后加载 tasks：

```
celery - test.celery worker -l debug #test 为 flask 脚本
```

然后启动 Flask 应用。关于 Flask 的项目应用的部署，可以使用 Nginx+Gunicorn 的方式去部署，这里只是简单演示就直接使用 python test.py 方式启动应用了。应用启动成功后，我们来写一个客户端 POST 脚本进行测试：

```

import requests
import json
import time
import sys
import argparse
import multiprocessing

ppm={'bjct': '1.2.3.4', 'hnct': '4.3.2.1'}

def tolist(fn):
    ips = []
    with open(fn) as f:

```



```

    for ip in f:
        ips.append(ip.strip())
    return ips

def kaitou(fn):
    with open(fn, 'r') as f:
        hosts=[ i.strip() for i in f.readlines()]
        idc_dict={}
        for i in hosts:
            key = i.split("-")[0]
            if key in idc_dict.keys():
                idc_dict[key].append(i)
            else:
                idc_dict[key]=[i]
                idc_dict[key].append(i)
        return idc_dict

def run(target, action, ips, users):
    p = {'ips': ips, 'users': users }
    r = requests.post('http://{0}:5000/{1}'.format(ppm[target], action),
        data = p)
    gto = r.json()[ 'goto' ]
    while 1:
        if requests.get("http://{0}:5000/{1}/result/{2}".
            format(ppm[target], action, gto)).json()[ 'state' ] ==
            "PENDING":
            print '\033[1;31;40m'
            print '*' * 50
            print target + ' IDC ' + "task running please wait....."
            print '*' * 50
            print '\033[0m'
            time.sleep(1)
            pass
            continue
        else:
            print '\033[1;32;40m'
            print target + ' IDC ' +
                "=====task running result===== "
            print '\033[0m'
            res=requests.get("http://{0}:5000/{1}/result/{2}".
                format(ppm[target], action, gto)).json()[ 'status' ]
            for i in res:

```

```

        print i + " " + str(res[i]).replace("u","")
    break

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--ips', help='ips files')
    parser.add_argument('-u', '--users', help='users files')
    parser.add_argument('-a', '--action', help='user manage action ex:
        add of del')
    args = vars(parser.parse_args())
    if args['ips'] and args['users'] and args['action'] in ['add','del'] :
        idcinfo=kaitou(args['ips'])
        idcs=idcinfo.keys()
        users=tolist(args['users'])
        pool = multiprocessing.Pool(processes=4)
        res=[]
        for idc in idcs:
            pool.apply_async(run, (idc,args['action'],idcinfo[idc],users))
        pool.close()
        pool.join()
    else:
        print parser.print_help()

```

因为我这个客户端脚本要实现一个简单的分布式功能，这个脚本的主要作用就是通过读取指定文件中的主机名地址和 user 用户列表，然后通过主机名的开头选择将任务发送到不同的 IDC 机房的 server 上。例如，如果我的主机文件内容如下：

```

(Ansible)[shencan@Ansible ~]$ cat y
bjct-mongodb01.shencan.net
hnct-rabbitmq01.shencan.net

```

使用 request 模块 POST 到 Flask 服务器，再根据返回的 celery tasks id 去查询状态，直到任务处理完，打印任务结果。我们运行的客户端脚本如下所示：

```

(Ansible)[shencan@Ansible ~]$ python test.py -i y -u z -a add
task running please wait.....

=====task running result=====
bjct-mongodb01.shencan.net {'failres': 0, 'skipped': 0, 'changed': 0,
    'ok': 1, 'nreachable': 0}
hnct-rabbitmq01.shencan.net {'failres': 0, 'skipped': 0, 'changed': 0,
    'ok': 1, 'nreachable': 0}

```

Flask 服务器的 Celery 日志如图 14-1 所示。

```
[2015-08-31 16:29:37,298: DEBUG/MainProcess] Task accepted: test.adduser[89c980ef-cb05-4572-95a4-3806a20d0dc1] pid:15268
[2015-08-31 16:29:37,341: WARNING/Worker-2] PLAY [all] *****
[2015-08-31 16:29:37,342: WARNING/Worker-2] TASK: [new user] *****
[2015-08-31 16:29:37,633: WARNING/Worker-2] ok: [10.10.138.24] => (item=a) => {
  "item": "a",
  "msg": "a"
}
[2015-08-31 16:29:37,648: WARNING/Worker-2] ok: [10.10.138.21] => (item=a) => {
  "item": "a",
  "msg": "a"
}
[2015-08-31 16:29:37,860: WARNING/Worker-2] ok: [10.10.138.24] => (item=b) => {
  "item": "b",
  "msg": "b"
}
[2015-08-31 16:29:37,870: WARNING/Worker-2] ok: [10.10.138.21] => (item=b) => {
  "item": "b",
  "msg": "b"
}
[2015-08-31 16:29:38,093: WARNING/Worker-2] ok: [10.10.138.24] => (item=c) => {
  "item": "c",
  "msg": "c"
}
[2015-08-31 16:29:38,095: WARNING/Worker-2] ok: [10.10.138.21] => (item=c) => {
  "item": "c",
  "msg": "c"
}
[2015-08-31 16:29:38,322: WARNING/Worker-2] ok: [10.10.138.24] => (item=d) => {
  "item": "d",
  "msg": "d"
}
[2015-08-31 16:29:38,390: WARNING/Worker-2] ok: [10.10.138.21] => (item=d) => {
  "item": "d",
  "msg": "d"
}
```

图 14-1 Flask 服务器的 Celery 日志

```
[2015-08-31 16:29:37,341: WARNING/Worker-2] PLAY [all] *****
[2015-08-31 16:29:37,342: WARNING/Worker-2] TASK: [new user] *****
[2015-08-31 16:29:37,633: WARNING/Worker-2] ok: [bjct-mongodb01.shencan.
net] => (item=a) => {
  "item": "a",
  "msg": "a"
}
[2015-08-31 16:29:37,648: WARNING/Worker-2] ok: [hnct-rabbitmq01.
shencan.net] => (item=a) => {
  "item": "a",
  "msg": "a"
}
[2015-08-31 16:29:37,860: WARNING/Worker-2] ok: [bjct-mongodb01.shencan.
net] => (item=b) => {
  "item": "b",
  "msg": "b"
}
[2015-08-31 16:29:37,870: WARNING/Worker-2] ok: [hnct-rabbitmq01.
shencan.net] => (item=b) => {
  "item": "b",
```

```

    "msg": "b"
  }
[2015-08-31 16:29:38,093: WARNING/Worker-2] ok: [bjct-mongodb01.shencan.
net] => (item=c) => {
  "item": "c",
  "msg": "c"
}
[2015-08-31 16:29:38,095: WARNING/Worker-2] ok: [hnct-rabbitmq01.
shencan.net] => (item=c) => {
  "item": "c",
  "msg": "c"
}
[2015-08-31 16:29:38,322: WARNING/Worker-2] ok: [bjct-mongodb01.shencan.
net] => (item=d) => {
  "item": "d",
  "msg": "d"
}
[2015-08-31 16:29:38,390: WARNING/Worker-2] ok: [hnct-rabbitmq01.
shencan.net] => (item=d) => {
  "item": "d",
  "msg": "d"
}

```

这里只是简单地使用 Ansible playbook API 调用了简单的 user.yaml playbook 文件，读者可以根据自己的实际情况去编写这个 playbook。

## 14.5 使用 jQuery Ajax 异步请求

在我们写 Web 平台的时候，如果提交一个任务运行时间过长，可能会导致 504 错误，我们已经使用 Celery 解决了这个问题。但是前端没法实时查看 Celery 的状态，这个时候就需要引入 jQuery Ajax 技术了。由于 jQuery Ajax 都是 Web 前端相关的技术，所以我这里只是简单介绍如何实现实时刷新的功能。

首先需要简单编写一个 HTML 页面，这里采用 GitHub 上 ansible-celery-flask-demo 这个项目的 HTML 页面中的 script 代码进行讲解：

```

<script>
  function playbook() {
    $('#progress').empty();

```

```

$.ajax({
    type: 'POST', // POST 方式
    url: '/add', // POST url
    data: $("#fuck").serialize(), // POST 数据 (把表单数据 id 为 fuck)
    success: function(data, status, request) {
        status_url = request.getResponseHeader('Location');
        // POST 成功后 Flask 会读取 Location 中带 task.id 的 URL
        update_progress(status_url); // 调用 update_progress 函数去查询 Celery 状态
    },
    error: function() { // POST 异常提示
        alert('Unexpected error');
    }
});

function update_progress(status_url) {
    $.getJSON(status_url, function(data) { // 调用 getJSON 请求 URL
        var txt = '';
        $.each(data.status, function(i, x) { // 循环分析 get 后的结果
            txt += x;
        });
        $('#progress').text(txt); // 将结果显示在 id 为 progress 处

        if (data['state'] == 'PENDING' || data['state'] == 'PROGRESS') { // 2 秒查询一次状态让判断 Celery task id 的状态
            setTimeout(function() { update_progress(status_url); }, 1000); // 上面条件不匹配继续调用 update_progress
        }
    });
}

$(function() {
    $('#start-bg-job').click(playbook); // 在 id 为 start-bg-job 处触发
});
</script>

```

需要注意的是，为了 Ajax POST 数据后 Flask 能返回含有 celery task id 的 URL，需要修改 Flask /add 路由的 return：



```
@app.route("/add",methods=['POST'])
def one():
    ips = [ i.encode('ascii') for i in request.form.getlist('ips') ]
    users = [ i.encode('ascii') for i in request.form.getlist('users') ]
    res = adduser.apply_async((ips, users))
    return jsonify({}), 202, {'Location': url_for('show_add_result',
        task_id=task.id)} # 将带 Celery id 的 URL 通过 HTTP Location header
        返回给 'Ajax'
```

下面我们来简单测试下。写一个简单的 HTML 页面，输入数据点击提交，如图 14-2 所示。

图 14-2 输入数据提交

执行任务后会马上返回 Pending... 状态，如图 14-3 所示。

图 14-3 执行任务后返回状态

Celery task 执行完成后会自动刷新显示结果，如图 14-4 所示。

图 14-4 显示结果

本节只是简单演示 Ajax 实现异步请求功能，读者可以参考以上思路在 Web 管理系统中实现相关功能。如果有前端开发经验的读者，可以通过一些 CSS 样式使得整个页面更加炫酷。

## 14.6 本章小结

本章主要围绕 Ansible 的 runner 和 playbook 两个常用 API 进行简单介绍，还介绍了一些简单的扩展。Ansible 软件本身的扩展性非常好，可以很方便地对它进行 2 次开发。为了能让 Ansible 能满足更加复杂的应用场景，本章还介绍一些如何使用 Web 框架实现 Ansible 远程调用 HTTP API，这样可以给想基于 Ansible 开发相关管理系统的读者提供一个更加好的思路。

## Ansible.cfg 配置文件参数详解

Ansible 的配置文件名称叫作 `ansible.cfg`，而默认 Ansible 程序读取配置 `ansible.cfg` 文件路径优先级如下：

- \* `ANSIBLE_CONFIG` (直接通过 `export` 变量声明)
- \* `ansible.cfg` (`ansible` 当前工作目录)
- \* `.ansible.cfg` (`ansible` 当前运行用户家目录)
- \* `/etc/ansible/ansible.cfg`

Ansible 会根据上面的优先级寻找配置文件，找到配置文件后会读取配置文件内的相应参数。`ansible.cfg` 是一个标准的 INI 格式文件，而且 Ansible 本身没有服务的概念，所以只要配置修改后配置将马上生效。在 Ansible 程序中有很多相关参数的定义，有些参数我们是需要通过修改配置文件来生效的。而有些参数配置我们可以直接通过 `bash` 环境变量进行临时更改。比如临时设置 Ansible 的资源清单文件为当前目录下的 `hosts` 文件：

```
export ANSIBLE_HOSTS=./hosts
```

下面将列出 Ansible 程序支持的所有环境变量更改的配置参数。大家也可以通过 Ansible 程序源码文件 `constants.py` 得到。所有以 `ANSIBLE_` 开头的参数都是可以通过 `bash` 环境变量进行临时修改的。由于篇幅有限下面只截取了部分配置：

```

DEFAULT_ACTION_PLUGIN_PATH      = get_config(p, DEFAULTS, 'action_
    plugins',          'ANSIBLE_ACTION_PLUGINS', '~/.ansible/plugins/action_
    plugins:/usr
r/share/ansible_plugins/action_plugins')
DEFAULT_CACHE_PLUGIN_PATH      = get_config(p, DEFAULTS, 'cache_
    plugins',          'ANSIBLE_CACHE_PLUGINS', '~/.ansible/plugins/cache_
    plugins:/usr/share/ansible_plugins/cache_plugins')
DEFAULT_CALLBACK_PLUGIN_PATH    = get_config(p, DEFAULTS, 'callback_
    plugins',          'ANSIBLE_CALLBACK_PLUGINS', '~/.ansible/plugins/
    callback_plugin
s:/usr/share/ansible_plugins/callback_plugins')
DEFAULT_CONNECTION_PLUGIN_PATH = get_config(p, DEFAULTS, 'connection_
    plugins', 'ANSIBLE_CONNECTION_PLUGINS', '~/.ansible/plugins/
    connection_plugins:/usr/share/ansible_plugins/connection_plugins')
DEFAULT_LOOKUP_PLUGIN_PATH      = get_config(p, DEFAULTS, 'lookup_
    plugins',          'ANSIBLE_LOOKUP_PLUGINS', '~/.ansible/plugins/lookup_
    plugins:/usr/share/ansible_plugins/lookup_plugins')
DEFAULT_VARS_PLUGIN_PATH        = get_config(p, DEFAULTS, 'vars_plugins',
    'ANSIBLE_VARS_PLUGINS', '~/.ansible/plugins/vars_plugins:/usr/share/
    ansible_plugins/vars_plugins')
DEFAULT_FILTER_PLUGIN_PATH      = get_config(p, DEFAULTS, 'filter_
    plugins',          'ANSIBLE_FILTER_PLUGINS', '~/.ansible/plugins/filter_
    plugins:/usr/share/ansible_plugins/filter_plugins')
CACHE_PLUGIN                    = get_config(p, DEFAULTS, 'fact_caching',
    'ANSIBLE_CACHE_PLUGIN', 'memory')
CACHE_PLUGIN_CONNECTION         = get_config(p, DEFAULTS, 'fact_caching_
    connection', 'ANSIBLE_CACHE_PLUGIN_CONNECTION', None)
CACHE_PLUGIN_PREFIX             = get_config(p, DEFAULTS, 'fact_caching_
    prefix', 'ANSIBLE_CACHE_PLUGIN_PREFIX', 'ansible_facts')
CACHE_PLUGIN_TIMEOUT            = get_config(p, DEFAULTS, 'fact_
    caching_timeout', 'ANSIBLE_CACHE_PLUGIN_TIMEOUT', 24 * 60 * 60,
    integer=True)
ANSIBLE_FORCE_COLOR             = get_config(p, DEFAULTS, 'force_color',
    'ANSIBLE_FORCE_COLOR', None, boolean=True)
ANSIBLE_NOCOLOR                 = get_config(p, DEFAULTS, 'nocolor',
    'ANSIBLE_NOCOLOR', None, boolean=True)
ANSIBLE_NOCOWS                 = get_config(p, DEFAULTS, 'nocows',
    'ANSIBLE_NOCOWS', None, boolean=True)
DISPLAY_SKIPPED_HOSTS          = get_config(p, DEFAULTS, 'display_
    skipped_hosts', 'DISPLAY_SKIPPED_HOSTS', True, boolean=True)
DEFAULT_UNDEFINED_VAR_BEHAVIOR = get_config(p, DEFAULTS, 'error_
    on_undefined_vars', 'ANSIBLE_ERROR_ON_UNDEFINED_VARS', True,
    boolean=True)

```

```

HOST_KEY_CHECKING          = get_config(p, DEFAULTS, 'host_key_
checking', 'ANSIBLE_HOST_KEY_CHECKING', True, boolean=True)
SYSTEM_WARNINGS            = get_config(p, DEFAULTS, 'system_
warnings', 'ANSIBLE_SYSTEM_WARNINGS', True, boolean=True)
DEPRECATION_WARNINGS      = get_config(p, DEFAULTS, 'deprecation_
warnings', 'ANSIBLE_DEPRECATION_WARNINGS', True, boolean=True)
DEFAULT_CALLABLE_WHITELIST = get_config(p, DEFAULTS, 'callable_
whitelist', 'ANSIBLE_CALLABLE_WHITELIST', [], islist=True)
COMMAND_WARNINGS          = get_config(p, DEFAULTS, 'command_
warnings', 'ANSIBLE_COMMAND_WARNINGS', False, boolean=True)
DEFAULT_LOAD_CALLBACK_PLUGINS = get_config(p, DEFAULTS, 'bin_ansible_
callbacks', 'ANSIBLE_LOAD_CALLBACK_PLUGINS', False, boolean=True)
DEFAULT_FORCE_HANDLERS    = get_config(p, DEFAULTS, 'force_
handlers', 'ANSIBLE_FORCE_HANDLERS', False, boolean=True)

```

下面以 Ansible 1.9 版本的默认的 ansible.cfg 进行解释。可以通过如下网址 <https://github.com/ansible/ansible/blob/stable-1.9/examples/ansible.cfg> 获取配置文件。配置文件内的大部分参数也可以通过 bash 环境变量的方式进行临时定义，变量名为 ANSIBLE\_ 加大写的配置参数，例如修改 Ansible 默认模块名称为 shell 模块：

```
export ANSIBLE_MODULE_NAME=shell
```

module\_name 就是 ansible.cfg 文件内的一个配置参数，只需要记住固定格式即可。

## defaults 配置块

配置项	注释	默认值
inventory	ansible inventory 文件路径	/etc/ansible/hosts
library	ansible 模块文件路径	无
remote_tmp	ansible 远程机器脚本临时存放目录	\$HOME/.ansible/tmp
Forks	ansible 执行并发数	5
poll_interval	ansible 异步任务查询间隔	15 秒
sudo_user	ansible sudo 用户	root
ask_sudo_pass	运行 ansible 是否提示输入 sudo 密码	True
ask_pass	运行 ansible 是否提示输入密码	True
transport	ansible 远程传输模式	smart
remote_port	远程主机 SSH 端口	22
module_lang	ansible 模块运行默认语言环境	C
gathering	facts 信息收集开关定义	Implicit (默认会收集)



(续)

配置项	注释	默认值
roles_path	ansible role 存放路径	/etc/ansible/roles
host_key_checking	SSH 主机 key 检测	False
sudo_exe	ansible sudo 执行命令	sudo
sudo_flags	ansible sudo 执行参数	-H
timeout	ansible SSH 连接超时时间	10 秒
remote_user	ansible 远程认证用户	root
log_path	ansible 日志记录文件	/var/log/ansible.log
module_name	ansible 默认执行模块	command
executable	ansible 命令执行 shell	/bin/sh
hash_behaviour	ansible 主机变量重复处理方式	replace
jinja2_extensions	jinja2 扩展列表	jinja2.ext.do jinja2.ext.i18n
private_key_file	ansible ssh 私钥文件	没定义
ansible_managed	在 jinja2 中 格式化 ansible_managed 变量	Ansible managed: {file} modified on %Y-%m-%d %H:%M:%S by {uid} on {host}
display_skipped_hosts	开启显示跳过的主机	True
error_on_undefined_vars	开启错误, 或者没有定义的变量	False
system_warnings	开启第三方包系统警告	True
action_plugins	ansible action 插件路径	/usr/share/ansible_plugins/action_plugins
callback_plugins	ansible callback 插件路径	/usr/share/ansible_plugins/callback_plugins
connection_plugins	ansible 连接插件路径	/usr/share/ansible_plugins/connection_plugins
lookup_plugins	ansible lookup 插件路径	插 /usr/share/ansible_plugins/lookup_plugins
vars_plugins	ansible vars 插件路径	/usr/share/ansible_plugins/vars_plugins
filter_plugins	ansible filter 插件路径	/usr/share/ansible_plugins/filter_plugins
bin_ansible_callbacks	开启 ansible 命令加载 callback 插件	False
Nocows	是否开启 ansiblenocows 图形	1
Nocolor	是否开启 ansible 颜色输出	1
http_user_agent	定义 ansible 请求 url UA	ansible-agent
fact_caching	定义 ansible facts 缓存方式	memory

## privilege\_escalation 配置块

配置项	注释	默认值
become	是否开启 become 模式	True
become_method	定义 become 方式	sudo
become_user	定义 become 用户	root
become_ask_pass	是否定义 become 提示密码	False

## paramiko\_connection 配置块

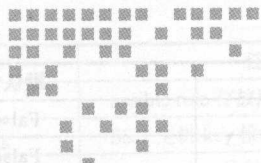
配置项	注释	默认值
record_host_keys	是否记录主机 key	False
pty	是否开启命令执行伪终端	False

## ssh\_connection 配置块

配置项	注释	默认值
ssh_args	定义 ansible ssh 参数	-o ControlMaster=auto -o ControlPersist=60s
control_path	ansible ssh 长连接控制文件	%(directory)s/ansible-ssh-%%h-%%p-%%r
pipelining	是否开启 pipelining 模式	False
scp_if_ssh	是否开启 scp 模式推送脚本	True

## accelerate 配置块

配置项	注释	默认值
accelerate_port	accelerate 远端监听端口	5099
accelerate_timeout	accelerate 模式, 超时时间	30 秒
accelerate_connect_timeout	accelerate 模式, 连接超时时间	5.0 秒
accelerate_daemon_timeout	accelerate 模式, demo 判断超时时间	30 秒
accelerate_multi_key	定义 accelerate 模式, 是否采用多 key 文件方	yes



## Appendix B

## 附录 B

# YAML 与 Jinja

我们使用 Ansible 的时候经常需要写 playbook 文件，在编写 playbook 之前我们需要了解下 playbook 文件的格式。Ansible 的 playbook 文件内容格式都是采用 YAML 标记语言去表达的，所以如果熟悉 YAML 标记语言对以后编写 playbook 会事半功倍。

在使用 Ansible 进行文件管理的时候，经常会采用以管理模板文件方式去管理，这样我们可以方便地维护一个模板文件去实现对不同需求的管理。Ansible 默认使用的模板引擎语言是 Jinja，这是一个功能强大的 Python 模板引擎语言。本附录将介绍 YAML 与 Jinja。

## YAML 标记语言

Ansible 默认会使用 PyYAML Python 库解析 playbook 的 YAML 文件内容。当然 YAML 标记语言还支持其他语言去解析。本附录只会围绕 PyYAML 库进行介绍。

### 基本语法例子

实例 1: # cat test.yaml

---

```

- 1
- 'two'
- ['three', 'four']
- {'five': 'six'}
# python
>>> import yaml
>>> print yaml.load(open('./test.yaml', "r").read())
[1, 'two', ['three', 'four'], {'five': 'six'}] # 打印结果

```

以上实例中 - 定义的值经过 PyYAML load 后转换为 Python 的 list 数据类型。

## 实例 2: # cat test.yaml

```

---
one: 1
two: 'two'
three: ['four', 'five']
six: {'seven': 'eight'}
# python
>>> import yaml
>>> print yaml.load(open('./test.yaml', "r").read())
{'six': {'seven': 'eight'}, 'three': ['four', 'five'], 'two': 'two',
'one': 1} # 打印结果

```

YAML 定义的 k/v 值对经过 PyYAML load 后转换为 Python 的 dict 数据类型。

下面的例子是对上面两种 YAML 标记形式进行汇总:

```

# cat test.yaml
---
one:
- 1
- 'two'
- ['three', 'four']
- {'five': 'six'}
two:
- 2
- 'two'
- ['three', 'four']
- {'five': 'six'}
# python
>>> import yaml

```

```
>>> print yaml.load(open('./test.yaml', "r").read())
{'two': [2, 'two', ['three', 'four']], {'five': 'six'}}, 'one': [1, 'two',
['three', 'four']], {'five': 'six'}} # 打印结果
```

## YAML 缩进符号

YAML 除了支持默认的缩进符号还支持 > 和 | 两种方式。这两种方式一般用于多行文本定义。定义的值经过 YAML load 之后都是 Python 的 str 数据类型。所以这两种缩进符号一般可以用于具有多行值的定义，例如：

```
# cat test.yaml
---
one: >
  1
two: >
  'two'
three: |
  ['four', 'five']
six: |
  {'seven': 'eight'}
# python

>>> import yaml
>>> print yaml.load(open('./test.yaml', "r").read())
{'one': '1\n', 'six': "{'seven': 'eight'}\n", 'three':
"['four', 'five']\n", 'two': "'two'\n"} # 打印结果
```

下面我们来看一个简单的例子，编写多行的 shell 命令，最后 Ansible 会把参数读作：

for i in {1..10}; do touch /tmp/\$i; done 的字符串进行运行。

```
# cat test.yaml
---
- hosts: all
  tasks:
    - name: test
      shell: >
        for i in {1..10};
        do
        touch /tmp/$i;
        done
```



主要我们需要了解 YAML 标记语言相关的变量定义以及转换到 Python 之后的数据类型。更多 PyYAML 相关的知识大家可以通过官网 (<http://pyyaml.org/wiki/PyYAMLDocumentation>) 进行学习和了解。

## Jinja 模板语言

Jinja 模板引擎在前面第 6 章“扩展 Ansible 组件”中已经介绍过，其实 Jinja 还有很多强大的功能，如果熟悉 Flask Web 框架，都会了解 Jinja 的一些功能，比如基本的 filter 和模板继承，以及 Jinja 的一些 for if 语法，等等。在使用 Ansible 的时候，一般会用到 Jinja 的 filter 以及一些 for 和 if 判断语法。关于 Jinja 的 filter 前面的章节已经介绍过了，本附录只介绍 Jinja 的一些常用语法。Jinja 的语法默认都是在 `{% %}` 包裹着。

### for 循环和 if 判断

Jinja 的 for 循环和 if 判断语法和 Python 语法一样：

```
{% set list= ['one', 'two', 'three'] %}
{% for i in list %}
    {{ i }}
{% endfor %}

-----

{% set list= ['one', 'two', 'three'] %}
{% for i in list %}
    {% if i == 'one' or i.startswith == 'two' %}
        -----> {{ i }}
    {% elif loop.index == 2 %}
        <-----> {{ i }}
    {% else %}
        <----- {{ i }}
    {% endif %}
{% endfor %}

-----

{% for key, value in dict.iteritems() %}
    {{ key }} ----> {{ value }}
{% endfor %}

-----
```

```
{% for dict in dicts %}
    {{ dict['key'] }}
{% endfor %}
```

Jinja 中可以使用 set 定义临时变量，也可以直接使用 Ansible 其他地方定义变量。关于 Jinja 变量的引用都是采用 {{ 变量名 }} 的方式，当然里面还可以根据变量名数据类型选择你想要的信息，比如 dict={'key': 'value'}，直接 {{ idct }} 会返回一个 python dict 数据，如果只需要 key 对应的值，则只需 {{ dict['key'] }} 或者 {{ dict.get('key') }}。其实这些都是 Python 的标准用法，在 Jinja 里面也可以直接使用。Python 的逻辑判断 and or not 在 Jinja 的判断中也可以直接使用。

## 空白控制

Jinja 默认 {% %} 定义的行渲染后都是空格，只需在 {% %} 中使用 - 符号即可控制空白输出：

```
{% set list= ['one', 'two', 'three'] %}
{% for i in list %}
    {{ i }}
{% endfor %}
```

比如上面的模板默认渲染出来的格式如下：

```
one
two
three
```

如果想实现在同一行显示，只需修改模板如下即可：

```
{% set list= ['one', 'two', 'three'] %}
{% for i in list -%}
    {{ i }}
{%- endfor %}
```

## 注释

Jinja 的注释也非常简单，直接使用 # 注释即可，但是 # 注释的地方要加在大括号 {} 内，例如：

```
{# note: disabled template because we no longer user this
  {% for user in users %}

      ...

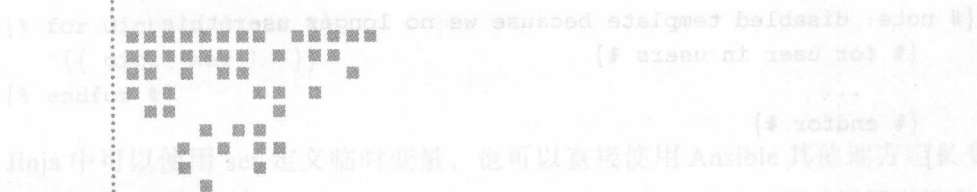
  {% endfor %}
#}
```

## 直接运算

Jinja 除了 filter 能对变量进行格式化或者加工，还支持直接对变量的值进行数学运算，比如想通过 facts 得到一台机器内存总和的一半，可以直接在 Jinja 模板通过 `{{ ansible_memtotal_mb/2 |int }}` 进行数学运算。当然 Jinja 还支持 `+` `-` `*` `**` `//` 数学运算。

## 其他高级功能

Jinja 还有很多其他高级功能，比如前面章节介绍的过滤器、模板继承、子模板、宏，因为这些功能在结合 Ansible 使用时并不常见，所以这里就不进行介绍了。感兴趣的读者可以通过官网 (<http://jinja.pocoo.org/docs/dev/>) 进行进一步学习。



## Appendix C

## 附录 C

# Ansible pull 模式

本书前面所有章节介绍的都是 Ansible 的 push 模式，这也是 Ansible 最常用的一种使用模式，在这种模式下读者经常会问，一台 Ansible 如果管理的目标机器过多的话，整个 Ansible 的效率是不是会很低。这个问题确实是存在，尽管你对 Ansible 进行一些简单的优化，但是如果每次操作目标机器数目过大的话，push 模式的效率就会显得很低。其实 Ansible 还有一种模式叫 pull 模式，pull 模式的所有操作命令都在 ansible-pull 命令下完成。这种模式类似于多台机器只针对自己本身进行配置管理。

## Ansible pull 模式流程

Ansible pull 模式运行需要一个 git 仓库，这个仓库会存放 Ansible pull 模式运行时所需的所有文件，包括 Ansible 的 inventory 和 playbook 文件。目标机器上会定时（用 crond）到 git 仓库内拉取文件，然后指定 inventory 文件运行相关的 playbook 文件。这个 playbook 文件一般只是针对本机进行配置管理，这里一般会采用 Ansible 的 local 方式进行本机管理，流程如图 C-1 所示。

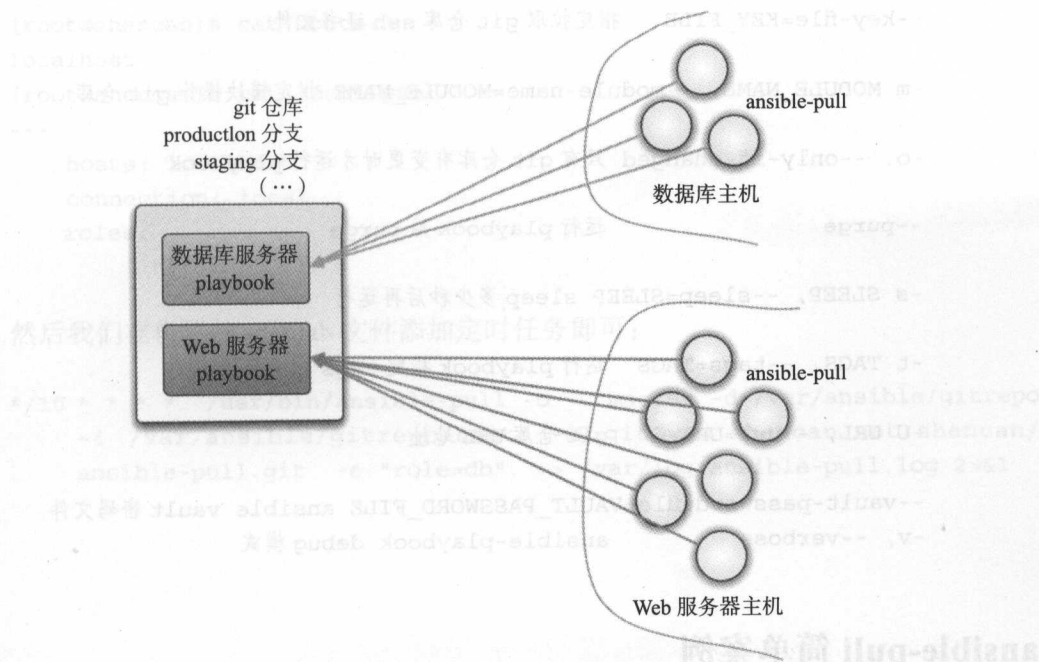


图 C-1 Ansible pull 模式流程图

## ansible-pull 命令参数

命令用法 : `ansible-pull [ 参数 ] [playbook.yml]`

参数 :

- `--accept-host-key`      接受 git 仓库 hostkey
- `-K, --ask-sudo-pass`      提示 sudo 密码
- `-C CHECKOUT, --checkout=CHECKOUT`      checkout 到 git 仓库指定分支或者 tags
- `-d DEST, --directory=DEST`      拉取 git 仓库到指定目录
- `-e EXTRA_VARS, --extra-vars=EXTRA_VARS`      传入变量
- `-f, --force`      强制运行 palybook ( 如果 git 仓库没有变更 )
- `-h, --help`      帮助信息
- `-i INVENTORY, --inventory-file=INVENTORY`      指定 inventory 文件



```

--key-file=KEY_FILE    指定拉取 git 仓库 ssh 证书文件

-m MODULE_NAME, --module-name=MODULE_NAME 指定模块操作 git 仓库

-o, --only-if-changed 只有 git 仓库有变更时才运行 playbook

--purge                运行 playbook 后 purge

-s SLEEP, --sleep=SLEEP sleep 多少秒后再运行

-t TAGS, --tags=TAGS   运行 playbook 指定 tags

-U URL, --url=URL       git 仓库 URL 地址

--vault-password-file=VAULT_PASSWORD_FILE ansible vault 密码文件

-v, --verbose           ansible-playbook debug 模式

```

## ansible-pull 简单案例

由于 Ansible pull 模式使用非常简单，只需要把 ansible-pull 命令以及相关参数放到定时任务即可。下面我们来简单看下 git 仓库目录结构，当然这里大家可以按照自己的需求去扩展这些 playbook，这里的 playbook 跟本书前面章节介绍的没有任何区别。

下面我们来看下 git 仓库里面的文件结构：

```
[root@shencan]# tree
```

```

.
├── hosts
├── local.yaml
├── roles
│   ├── db
│   │   ├── files
│   │   ├── tasks
│   │   └── templates
│   └── web
│       ├── files
│       ├── tasks
│       └── templates

```

```
9 directories, 2 files
```

```
[root@shencan]# cat hosts
localhost
[root@shencan]# cat local.yaml
```

```
---
hosts: localhost
connection: local
roles:
  - "{{ role }}"
```

然后我们在机器的 crontab 文件添加定时任务即可：

```
* /10 * * * * /usr/bin/ansible-pull -o -C master -d /var/ansible/gitrepo
-i /var/ansible/gitrepo/hosts -U git@www.shencan.net:shencan/
ansible-pull.git -e "role=db" >> /var/log/ansible-pull.log 2>&1
```

## SSH Forward 模式

在绝大数互联网公司为了统一管理服务器登录以及相关权限管理，都会采用通过跳板机方式登录服务器，而且服务器必须通过跳板机才能登录。运维人员经常要管理一大批机器，而且跳板机是公共服务设备，所以每个人的权限都被束缚着，特别是很多 Mac 系统和 Ubuntu 系统用户，喜欢用自己的环境维护一些项目。这个时候想直接通过本机直接管理线上的服务器，由于服务器受到跳板机的登录限制，导致我们不能直接通过本机电脑管理目标服务器。很多公司不同的 IDC 机房之间也做了类似的安全限制，比如你想登录某个 IDC 机房的服务器就必须得先登录到该 IDC 机房内的一台跳板机，然后才能登录服务器。这种多层网络架构对于配置管理工具而言，确实不好直接进行管理，但是都可以通过相关代理技术或者软件自身多层架构进行管理。例如可以使用 Nginx 代理 Puppet 服务器实现夸机房管理，而 SaltStack 软件自带 Syndic 多层网络架构解决方案。不过 Ansible 默认没有类似的解决方案。

因为 Ansible 默认使用 SSH 服务进行远程管理，如果想要 Ansible 能实现跨机房的解决方案，首先我们得想办法解决 SSH 服务是否可行。其实 SSH 自带一种叫 Agent Forward 模式，如图 D-1 所示，我需要通过 Jumpbox 的 Agent Forward 模式管理 hosts 主机。下面我们通过一个实例来配置 SSH 的 Forward 模式。

配置很简单只需要在 Mac 系统上修改 SSH 客户端配置文件，然后保证 Mac 系统免密登录 Jumpbox 机器：

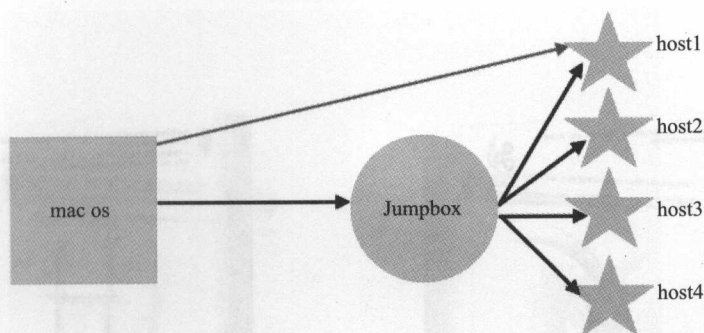


图 D-1 Forward 模式

```

Host *
Protocol 2
Host 10.64.115.27
    ForwardAgent      yes
    ProxyCommand       ssh shencan@10.0.0.1 nc %h %p
    User               shencan
    IdentityFile       /Users/shencan/key/jumpbox.key
  
```

上面的配置表示 10.64.115.27 是远程主机，10.0.0.1 是 Jumpbox 机器。通过 shencan 用户免密连接 10.0.0.1，然后 Jumpbox 通过 shencan 用户的 /Users/shencan/key/jumpbox.key 私钥文件登录 10.64.115.27 主机。

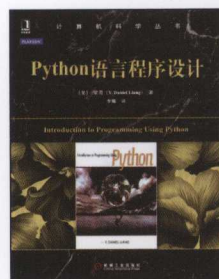
下面需要 Ansible 程序引用这个 SSH 客户端文件。修改 ansible.cfg 中的 ssh\_connection 配置项的 ssh\_args 的参数：

```
ssh_args = -o ControlMaster=auto -o ControlPersist=60s -F /Users/shencan/.ssh/config
```

-F 表示引用上面定义的 SSH 客户端配置文件，然后我们指定目标主机运行 Ansible 即可：

```

ansible -i hosts 10.64.115.27 -u shencan -a 'df -h'
10.64.115.27 | success | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2       3.6T  415G   3.2T  12% /
tmpfs           63G      0    63G   0% /dev/shm
  
```





Ansible是目前业界最火的自动化运维工具。与Puppet等DevOps工具不同的是，Ansible的无客户端部署模式，极大地降低了用户门槛，而其优良的设计又不失配管编排的架构优势。本书内容由浅入深，方便读者掌握，并快速运用到自己的实际环境中。跟随着本书的章节递进，读者可以逐渐掌握到大规模环境下的自动化运维能力，真正成为DevOps的弄潮儿。

—— 日志易产品总监，前微博系统架构师 饶琛琳

本书作者之一灿哥热衷于运维事业，是国内早期的运维自动化发起人，灿哥运维技术学识渊博、平时注意积累，博客的文章被广为流传。一直期待Ansible相关书籍，这本书将成为运维领域的经典之作。

—— 搜狐社交产品中心Python服务端负责人 张斌

与灿哥相识已久，多年来他一直专注于自动化运维的研究，并长期奋战在国内知名互联网公司，积累了丰富的经验。此书是灿哥又一倾力之作，也是国内工程师写的第一本Ansible书籍，可谓干货满满，值得细细研读。

—— 优酷土豆公司高级信息安全工程师 徐元振

